O'REILLY®

原书第2版

利用Python进行数据分析

Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython



powered by jupyter

Wes McKinney 著

Sean Cheney 译

VISIT…

LANZAROTE Caliente.COM

利用 Python 进行数据分析 中文第二版

译者：SeanCheney

来源：https://www.jianshu.com/c/52882df3377a

下载本书：http://www.jianshu.com/p/fad9e41c1a42（更新为GitHub链接）
下载本书代码：https://github.com/wesm/pydata-book（建议把代码下载下来之后，安装好Anaconda 3.6，在目录文件夹中用Jupyter notebook打开）

本书是2017年10月20号正式出版的，和第1版的不同之处有：

- 包括Python教程内的所有代码升级为Python 3.6（第1版使用的是Python 2.7）
- 更新了Anaconda和其它包的Python安装方法
- 更新了Pandas为2017最新版
- 新增了一章，关于更高级的Pandas工具，外加一些tips
- 简要介绍了使用StatsModels和scikit-learn

对有些内容进行了重新排版。

译者注1：最大的改变是把第1版附录中的Python教程，单列成了现在的第2章和第3章，并且进行了扩充。可以说，本书第2版对新手更为友好了！

译者注2：毫无疑问，本书是学习Python数据分析最好的参考书。本来想把书名直接译为《Python数据分析》，这样更简短。但是为了尊重第1版的翻译，考虑到继承性，还是用老书名。这样读过第一版的老读者可以方便的用之前的书名检索到第二版。作者在写第二版的时候，有些文字是照搬第一版的。所以第二版的翻译也借鉴copy了第一版翻译：即，如果第二版中有和第一版相同的文字，则copy第一版的中文译本，觉得不妥的地方会稍加修改，剩下的不同的内容就自己翻译。这样做也是为读过第一版

的老读者考虑——相同的内容可以直接跳过。

目录

前言	1
第 1 章 准备工作	9
第 2 章 Python 语言基础、IPython 及 Jupyter notebook	24
第 3 章 内建数据结构、函数及文件	68
第 4 章 NumPy 基础：数组与向量化计算	114
第 5 章 pandas 入门	166
第 6 章 数据载入、存储及文件格式	225
第 7 章 数据清洗与准备	257
第 8 章 数据规整：连接、联合与重塑	297
第 9 章 绘图与可视化	339
第 10 章 数据聚合与分组操作	376
第 11 章 时间序列	417
第 12 章 高阶 pandas	479
第 13 章 Python 建模库介绍	504
第 14 章 数据分析示例	530
附录 A 高阶 NumPy	593
附录 B 更多 IPython 系统相关内容	638

前言

第 2 版新内容

本书第 1 版出版于 2012 年，彼时基于 Python 的开源数据分析库（例如 pandas）仍然是一个发展迅速的新事物。在本次更新、拓展的第 2 版中，我在一些章节内进行了修改，以解释过去 5 年中发生的不兼容的变更、弃用和一些新特性。此外，我还添加了新内容，用以介绍在 2012 年还不存在或者不成熟的工具。最后，我会避免把一些新兴的或者不太可能走向成熟的开源项目写入本书。我希望本版的读者能够发现本书内容在 2020 年或者 2021 年仍然几乎像在 2017 年一样适用。

第 2 版中的主要更新包括：

所有的代码，包括把 Python 的教程更新到了 Python 3.6 版本（第 1 版中使用的是 Python 2.7）

更新了 Python 第三方发布版 Anaconda 和其他所需 Python 包的安装指引

更新 pandas 库到 2017 年的最新版

新增一章，关于更多高级 pandas 工具和一些使用提示

新增 statsmodels 和 scikit-learn 的简明使用介绍

除了以上更新内容，我还重新组织了第 1 版的部分重要内容，使本书对新手来说更易于理解。

本书约定

以下印刷约定将在本书中使用：

斜体（Italic）

表示新的术语、URL、email 地址、文件名和文件扩展名。

等宽字体（Constant width）

用于程序清单以及段落中的程序元素，例如变量名、函数名、数据库、数据类型、环境变量、表达式和关键字等。

等宽粗体（Constant width bold）

表示命令或其他应当由用户键入的文本。

等宽斜体（Constant width italic）

表示应当由用户提供的值来替代的文本，或者其他由上下文决定的值。

本符号表示提示或建议。

本符号表示一般性说明。

本符号表示警告。

使用代码示例

可以通过本书的 GitHub 仓库获取本书每一章中的数据文件和相关材料。GitHub 仓库地址：http://github.com/wesm/pydata-book。

本书的目的在于帮助你完成工作。一般来说，本书提供的示例代码，你可以在你的程序或文档中使用而无须联系我们获取许可，除非你需要重造大量代码。举例来说，使用本书中的代码段编写程序无须授权许可，但销售或发行 O 拎 eilly 图书的 CD-ROM 代码示例则需要许可。引用本书代码回答问题不需要许可，但在你的产品文档中大量使用本书示例代码则需要许可。

我们鼓励注明资料来源的行为，但这并不是必需的。来源注明通常包括书名、作者、出版社及 ISBN，例如："Python for Data Analysis by Wes McKinney(O 拎 eilly). Copyright 2017 Wes McKinney, 978-1-491-95766-0"。

如果你认为你对本书示例代码的使用超过了正常使用范围或者需要以上介绍的授权许可，请联系 permissions@oreilly.com。

O'Reilly Safari

Safari（前身为 Safari Books Online ）是一个会员制的培训、参考网站，服务于企业、政府、教育者和个人。

会员可以访问数千书籍、培训视频、学习路径、交互教程和超过 250 家出版商的企划列表，包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、 Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。

更多信息，请访问 http://oreilly.com/safari。

如何联系我们

对于本书如果有任何意见或疑问，请按照以下地址联系本书出版商。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

我们为本书准备了一个网页，用于陈列勘误、示例和其他附加信息。访问地址是：http://bit.ly/python_data_analysis_2e。

针对本书评论或提出技术问题，请发送邮件至：bookquestions@oreilly.com。

关于本书的更多信息、课程、会议及新闻，请访问我们的网站：http://www.oreilly.com。

Facebook 联系我们：http://facebook.com/oreilly

Twitter 联系我们：http://twitter.com/oreillymedia

YouTube 观看我们的视频：http://www.youtube.com/oreillymedia

致谢

本书是全世界很多人多年来富有成效的讨论、协作和支持的成果。我想对他们中的一些代表致以谢意。

怀念：John D. Hunter（1968—2012）

我们亲爱的朋友和同行 John D. Hunter 在经历了一场与结肠癌的战斗后，于 2012 年 8 月 28 日离开了世界。那时正是我完成本书第 1 版最终手稿后不久。

John 对 Python 科学计算和数据社区的影响之大难以估量，他给我们留下的遗产价值非凡。除了在 2000 年初期开发 matplotlib 之外（那时

Python 还没有当下如此流行），他还帮助塑造了一代核心开源开发者的文化，如今这些开发者已经成为 Python 生态系统的顶梁柱，而 Python 生态系统对于现如今的我们来说似乎是理所当然的。

在 2010 年 1 月，我开源生涯的早期，那时候 pandas 刚刚发布了 0.1 版本，我便有幸结识了 John。即便在最黑暗的时期，他的才华和指导仍在帮助我推动 pandas 前进，实现 Python 成为数据分析第一语言的愿景。

John 与 IPython、Jupyter 项目的先锋 Fernando Pérez、Brian Granger 及其他很多 Python 社区的倡议人联系紧密。我们四人曾经希望共同写作一本书，但只有我个人时间最为自由，所以这个想法被搁置了。我非常确信他会为过去 5 年中我们个人及我们社区所取得的成就感到骄傲。

第 2 版致谢（2017）

距离我在 2012 年 7 月完成第 1 版手稿已经 5 年了。很多事情都发生了变化。Python 社区获得了极大的成长，围绕 Python 的开源软件生态系统也十分繁荣。pandas 核心开发者孜孜不倦的付出，使得 pandas 项目高速成长，也使得 pandas 的用户群体遍布 Python 数据科学生态系统的各个角落，没有他们本书将不会存在。pandas 的核心开发者包括但不限于：Tom Augspurger、Joris van den Bossche、Chris Bartak、Phillip Cloud、gfyoung、Andy Hayden、Masaaki Horikoshi、Stephan Hoyer、Adam Klein、Wouter、Overmeire、Jeff Reback、Chang She、Skipper Seabold、Jeff Tratner 和 y-p。

在第 2 版的实际写作过程中，非常感谢 O 拎 eilly 的工作人员在写作进程中给予的耐心帮助。他们是 Marie Beaugureau、Ben Lorica 和 Colleen Toporek。我再次得到了优秀技术审阅人的支持，他们是 Tom Augpurger、Paul Barry、Hugh Brown、Jonathan Coe 和 Andreas Müller。感谢你们。

本书的第 1 版已经被翻译成多种语言，包括汉语、法语、德语、日语、韩语和俄语。将本书翻译给外国读者，是一份工作量大且缺少关注的付出。感谢你们帮助全世界更多人士学会如何编程及使用数据分析工具。

在过去几年中，Cloudera 和 Two Sigma 投资公司对我的持续开源开发工作的支持使我感到十分幸运。由于开源项目相对于用户基数的比例越来越小，向重要开源项目提供开发支持变得越来越重要。这是一件值得去做的正确工作。

第 1 版致谢（2012）

如果没有众多相关人士的支持，写作本书对我来说将会十分困难。

对于 O 拎 eilly 的工作人员，我非常感谢我的编辑 Meghan Blanchette 和 Julie Steele，他们在整个写作过程对我给予指导。Mike Loukides 还在建议阶段与我一起工作，帮助本书付梓。

我收到了大量相关人士丰富的技术审阅。尤其是 Martin Blais 和 Hugh Brown ，他们自始至终在提高本书示例的清晰度、组织度上提供了令人难以置信的帮助。James Long、Drew Conway、Fernando Pérez、Brian Granger、Thomas Kluyver、Adam Klein、Josh Klein、Chang She

和 Stéfan van der Walt 每个人都审阅了本书的一章或多章，从很多角度提供了有效反馈。

我从数据社区的朋友和同行那里获得了很多关于示例和数据集的优秀想法，他们是：Mike Dewar、Jeff Hammerbacher、James Johndrow、Kristian Lum、Adam Klein、Hilary Mason、Chang She 和 Ashley Williams。

当然，我也非常感激开源科学 Python 社区的众多领头人，他们为我的开发工作打下了基础，并在本书写作过程中给予我鼓励：IPython 核心团队（Fernando Pérez、Brian Granger、Min Ragan-Kelly、Thomas Kluyver 和其他相关人士）、John Hunter、Skipper Seabold、Travis Oliphant、Peter Wang、Eric Jones、Robert Kern、Josef Perktold、Francesc Alted、Chris Fonnesbeck 以及其他由于人数太多而无法提及的人们。还有很多人士一直以来提供了支持、想法和鼓励：Drew Conway、Sean Taylor、Giuseppe Paleologo、Jared Lander、David Epstein、John Krowas、Joshua Bloom、Den Pilsworth、John Myles-White 以及很多已经忘了姓名的人士。

我还要感谢前些年生活中的一些人。首先是在 AQR 的前同事，他们都曾为我在 pandas 方面的工作喝彩，他们是：Alex Reyfman、Michael Wong、Tim Sargen、Oktay Kurbanov、Matthew Tschantz、Roni Israelov、Michael Katz、Chris Uga、Prasad Ramanan、Ted Square 和 Hoon Kim，以及我的指导教授 Haynes Miller（麻省理工学院）和 Mike West（杜克大学）。

2014 年，我在更新本书代码示例、修正一些由于 pandas 变更产生的错误时，从 Phillip Cloud 和 Joris Van den Bossche 处获得了重要帮助。

个人方面，感谢 Casey 在写作过程中为我提供了无价的日常支持，忍受我的情绪起伏直到我按计划表写出了最终手稿。最后，感谢我的父母 Bill 和 Kim，他们教会我如何去追寻梦想、永不止步。

第1章 准备工作

1.1 本书的内容

本书讲的是利用Python进行数据控制、处理、整理、分析等方面的具体细节和基本要点。我的目标是介绍Python编程和用于数据处理的库和工具环境，掌握这些，可以让你成为一个数据分析专家。虽然本书的标题是“数据分析”，重点确实Python编程、库，以及用于数据分析的工具。这就是数据分析要用到的Python编程。

什么样的数据？

当书中出现“数据”时，究竟指的是什么呢？主要指的是结构化数据（structured data），这个故意含糊其辞的术语代指了所有通用格式的数据，例如：

- 表格型数据，其中各列可能是不同的类型（字符串、数值、日期等）。比如保存在关系型数据库中或以制表符/逗号为分隔符的文本文件中的那些数据。
- 多维数组（矩阵）。
- 通过关键列（对于SQL用户而言，就是主键和外键）相互联系的多个表。
- 间隔平均或不平均的时间序列。

这绝不是一个完整的列表。大部分数据集都能被转化为更加适合分析和建模的结构化形式，虽然有时这并不是很明显。如果不行的话，也可以将数据集的特征提取为某种结构化形式。例如，一组新闻文章可以被处理为一张词频表，而这张词频表就可以用于情感分析。

大部分电子表格软件（比如Microsoft Excel，它可能是世界上使用最广泛的数据分析工具了）的用户不会对此类数据感到陌生。

1.2 为什么要使用Python进行数据分析

许许多多的人（包括我自己）都很容易爱上Python这门语言。自从1991年诞生以来，Python现在已经成为最受欢迎的动态编程语言之一，其他还有Perl、Ruby等。由于拥有大量的Web框架（比如Rails（Ruby）和Django（Python）），自从2005年，非常流行使用Python和Ruby进行网站建设工作。这些语言常被称作脚本（scripting）语言，因为它们可以用于编写简短而粗糙的小程序（也就是脚本）。我个人并不喜欢“脚本语言”这个术语，因为它好像在说这些语言无法用于构建严谨的软件。在众多解释型语言中，由于各种历史和文化的原因，Python发展出了一个巨大而活跃的科学计算（scientific computing）社区。在过去的10年，Python从一个边缘或“自担风险”的科学计算语言，成为了数据科学、机器学习、学界和工业界软件开发最重要的语言之一。

在数据分析、交互式计算以及数据可视化方面，Python将不可避免地与其他开源和商业的领域特定编程语言/工具进行对比，如R、MATLAB、SAS、Stata等。近年来，由于Python的库（例如pandas和scikit-learn）不断改良，使其成为数据分析任务的一个优选方案。结合其在通用编程方面的强大实力，我们完全可以只使用Python这一种语言构建以数据为中心的应用。

Python作为胶水语言

Python能变为成功的科学计算工具的部分原因是，它能够轻松地集成C、C++以及Fortran代码。大部分现代计算环境都利用了一些Fortran和C库来实现线性代数、优选、积分、快速傅里叶变换以及其他诸如此类的算法。许多企业和国家实验室也利用Python来“粘合”那些已经用了多年的遗留软件系统。

大多数软件都是由两部分代码组成的：少量需要占用大部分执行时间的代码，以及大量不经常执行的“胶水代码”。大部分情况下，胶水代码的执行时间是微不足道的。开发人员的精力几乎都是花在优化计算瓶颈上面，有时更是直接转用更低级的语言（比如C）。

解决“两种语言”问题

很多组织通常都会用一种类似于领域特定的计算语言（如SAS和R）对新的想法进行研究、原型构建和测试，然后再将这些想法移植到某个更大的生产系统中去（可能是用Java、C#或C++编写的）。人们逐渐意识到，Python不仅适用于研究和原型构建，同时也适用于构建生产系统。为什么一种语言就够了，却要使用两个语言的开发环境呢？我相信越来越多的企业也会这样看，因为研究人员和工程技术人员使用同一种编程工具将会给企业带来非常显著的组织效益。

为什么不选Python

虽然Python非常适合构建分析应用以及通用系统，但它对不少应用场景适用性较差。

由于Python是一种解释型编程语言，因此大部分Python代码都要比用编译型语言（比如Java和C++）编写的代码运行慢得多。由于程序员的时间通常都比CPU时间值钱，因此许多人也愿意在这里做一些权衡。但是，在那些要求延迟非常小或高资源利用率的应用中（例如高频交易系统），耗费时间使用诸如C++这样更低级、更低生产率的语言进行编程也是值得的。

对于高并发、多线程的应用程序而言（尤其是拥有许多计算密集型线程的应用程序），Python并不是一种理想的编程语言。这是因为Python有一个叫做全局解释器锁（Global Interpreter Lock，GIL）的组件，这是一种防止解释器同时执行多条Python字节码指令的机制。有关“为什么会存在GIL”的技术性原因超出了本书的范围。虽然很多大数据处理应用程序为了能在较短的时间内完成数据集的处理工作都需要运行在计算机集群上，但是仍然有一些情况需要用单进程多线程系统来解决。

这并不是说Python不能执行真正的多线程并行代码。例如，Python的C插件使用原生的C或C++的多线程，可以并行运行而不被GIL影响，只要它们不频繁地与Python对象交互。

1.3 重要的Python库

考虑到那些还不太了解Python科学计算生态系统和库的读者，下面我先对各个库做一个简单的介绍。

NumPy

NumPy（Numerical Python的简称）是Python科学计算的基础包。本书大部分内容都基于NumPy以及构建于其上的库。它提供了以下功能（不限于此）：

- 快速高效的多维数组对象ndarray。
- 用于对数组执行元素级计算以及直接对数组执行数学运算的函数。
- 用于读写硬盘上基于数组的数据集的工具。
- 线性代数运算、傅里叶变换，以及随机数生成。
  -成熟的C API， 用于Python插件和原生C、C++、Fortran代码访问NumPy的数据结构和计算工具。

除了为Python提供快速的数组处理能力，NumPy在数据分析方面还有另外一个主要作用，即作为在算法和库之间传递数据的容器。对于数值型数据，NumPy数组在存储和处理数据时要比内置的Python数据结构高效得多。此外，由低级语言（比如C和Fortran）编写的库可以直接操作NumPy数组中的数据，无需进行任何数据复制工作。因此，许多Python的数值计算工具要么使用NumPy数组作为主要的数据结构，要么可以与NumPy进行无缝交互操作。

pandas

pandas提供了快速便捷处理结构化数据的大量数据结构和函数。自从2010年出现以来，它助使Python成为强大而高效的数据分析环境。本书用得最多的pandas对象是DataFrame，它是一个面向列（column-oriented）的二维表结构，另一个是Series，一个一维的标签化数组对象。

pandas兼具NumPy高性能的数组计算功能以及电子表格和关系型数据库（如SQL）灵活的数据处理功能。它提供了复杂精细的

索引功能，以便更为便捷地完成重塑、切片和切块、聚合以及选取数据子集等操作。因为数据操作、准备、清洗是数据分析最重要的技能，pandas是本书的重点。

作为一点背景，我是在2008年初开始开发pandas的，那时我任职于AQR Capital Management，一家量化投资管理公司，我有许多工作需求都不能用任何单一的工具解决：

- 有标签轴的数据结构，支持自动或清晰的数据对齐。这可以防止由于数据不对齐，和处理来源不同的索引不同的数据，造成的错误。
- 集成时间序列功能。
- 相同的数据结构用于处理时间序列数据和非时间序列数据。
- 保存元数据的算术运算和压缩。
- 灵活处理缺失数据。
- 合并和其它流行数据库（例如基于SQL的数据库）的关系操作。

我想只用一种工具就实现所有功能，并使用通用软件开发语言。Python是一个不错的候选语言，但是此时没有集成的数据结构和工具来实现。我一开始就是想把pandas设计为一款适用于金融和商业分析的工具，pandas专注于深度时间序列功能和工具，适用于时间索引化的数据。

对于使用R语言进行统计计算的用户，肯定不会对DataFrame这个名字感到陌生，因为它源自于R的data.frame对象。但与Python不同，data frames是构建于R和它的标准库。因此，pandas的许多功能不属于R或它的扩展包。

pandas这个名字源于panel data（面板数据，这是多维结构化数据集在计量经济学中的术语）以及Python data analysis（Python数据分析）。

matplotlib

matplotlib是最流行的用于绘制图表和其它二维数据可视化的

Python库。它最初由John D.Hunter（JDH）创建，目前由一个庞大的开发人员团队维护。它非常适合创建出版物上用的图表。虽然还有其它的Python可视化库，matplotlib却是使用最广泛的，并且它和其它生态工具配合也非常完美。我认为，可以使用它作为默认的可视化工具。

IPython和Jupyter

IPython项目起初是Fernando Pérez在2001年的一个用以加强和Python交互的子项目。在随后的16年中，它成为了Python数据栈最重要的工具之一。虽然IPython本身没有提供计算和数据分析的工具，它却可以大大提高交互式计算和软件开发的生产率。IPython鼓励“执行-探索”的工作流，区别于其它编程软件的“编辑-编译-运行”的工作流。它还可以方便地访问系统的shell和文件系统。因为大部分的数据分析代码包括探索、试错和重复，IPython可以使工作更快。

2014年，Fernando和IPython团队宣布了Jupyter项目，一个更宽泛的多语言交互计算工具的计划。IPython web notebook变成了Jupyter notebook，现在支持40种编程语言。IPython现在可以作为Jupyter使用Python的内核（一种编程语言模式）。

IPython变成了Jupyter庞大开源项目（一个交互和探索式计算的高效环境）中的一个组件。它最老也是最简单的模式，现在是一个用于编写、测试、调试Python代码的强化shell。你还可以使用通过Jupyter Notebook，一个支持多种语言的交互式网络代码“笔记本”，来使用IPython。IPython shell 和Jupyter notebooks特别适合进行数据探索和可视化。

Jupyter notebooks还可以编写Markdown和HTML内容，提供了一种创建代码和文本的富文本方法。其它编程语言也在Jupyter中植入了内核，好让在Jupyter中可以使用Python另外的语言。

对我个人而言，我的大部分Python都要用到IPython，包括运行、调试和测试代码。

在本书的GitHub页面，你可以找到包含各章节所有代码实例的Jupyter notebooks。

SciPy

SciPy是一组专门解决科学计算中各种标准问题域的包的集合，主要包括下面这些包：

- scipy.integrate：数值积分例程和微分方程求解器。
- scipy.linalg：扩展了由numpy.linalg提供的线性代数例程和矩阵分解功能。
- scipy.optimize：函数优化器（最小化器）以及根查找算法。
- scipy.signal：信号处理工具。
- scipy.sparse：稀疏矩阵和稀疏线性系统求解器。
- scipy.special：SPECFUN（这是一个实现了许多常用数学函数（如伽玛函数）的Fortran库）的包装器。
- scipy.stats：标准连续和离散概率分布（如密度函数、采样器、连续分布函数等）、各种统计检验方法，以及更好的描述统计法。

NumPy和SciPy结合使用，便形成了一个相当完备和成熟的计算平台，可以处理多种传统的科学计算问题。

scikit-learn

2010年诞生以来，scikit-learn成为了Python的通用机器学习工具包。仅仅七年，就汇聚了全世界超过1500名贡献者。它的子模块包括：

- 分类：SVM、近邻、随机森林、逻辑回归等等。
- 回归：Lasso、岭回归等等。
- 聚类：k-均值、谱聚类等等。
- 降维：PCA、特征选择、矩阵分解等等。
- 选型：网格搜索、交叉验证、度量。
- 预处理：特征提取、标准化。

与pandas、statsmodels和IPython一起，scikit-learn对于Python成为高效数据科学编程语言起到了关键作用。虽然本书不会详细讲解scikit-learn，我会简要介绍它的一些模型，以及用其它工具如何使用这些模型。

statsmodels

statsmodels是一个统计分析包，起源于斯坦福大学统计学教授Jonathan Taylor，他设计了多种流行于R语言的回归分析模型。Skipper Seabold和Josef Perktold在2010年正式创建了statsmodels项目，随后汇聚了大量的使用者和贡献者。受到R的公式系统的启发，Nathaniel Smith发展出了Patsy项目，它提供了statsmodels的公式或模型的规范框架。

与scikit-learn比较，statsmodels包含经典统计学和经济计量学的算法。包括如下子模块：

- 回归模型：线性回归，广义线性模型，健壮线性模型，线性混合效应模型等等。
- 方差分析（ANOVA）。
- 时间序列分析：AR，ARMA，ARIMA，VAR和其它模型。
- 非参数方法： 核密度估计，核回归。
- 统计模型结果可视化。

statsmodels更关注与统计推断，提供不确定估计和参数p-值。相反的，scikit-learn注重预测。

同scikit-learn一样，我也只是简要介绍statsmodels，以及如何用NumPy和pandas使用它。

1.4 安装和设置

由于人们用Python所做的事情不同，所以没有一个普适的Python及其插件包的安装方案。由于许多读者的Python科学计算环境都不能完全满足本书的需要，所以接下来我将详细介绍各个操作系统上的安装方法。我推荐免费的Anaconda安装包。写作本书

时，Anaconda提供Python 2.7和3.6两个版本，以后可能发生变化。本书使用的是Python 3.6，因此推荐选择Python 3.6或更高版本。

Windows

要在Windows上运行，先下载Anaconda安装包。推荐跟随Anaconda下载页面的Windows安装指导，安装指导在写作本书和读者看到此文的的这段时间内可能发生变化。

现在，来确认设置是否正确。打开命令行窗口（`cmd.exe`），输入`python`以打开Python解释器。可以看到类似下面的Anaconda版本的输出：

```
C:\Users\wesm>python
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul
[MSC v.1900 64 bit (AMD64)] on win32
>>>
```

要退出shell，按Ctrl-D（Linux或macOS上），Ctrl-Z（Windows上），或输入命令`exit()`，再按Enter。

Apple (OS X, macOS)

下载OS X Anaconda安装包，它的名字类似Anaconda3-4.1.0-MacOSX-x86_64.pkg。双击.pkg文件，运行安装包。安装包运行时，会自动将Anaconda执行路径添加到`.bash_profile`文件，它位于`/Users/$USER/.bash_profile`。

为了确认成功，在系统shell打开IPython：

```
$ ipython
```

要退出shell，按Ctrl-D，或输入命令`exit()`，再按Enter。

GNU/Linux

Linux版本很多，这里给出Debian、Ubantu、CentOS和Fedora的安装方法。安装包是一个脚本文件，必须在shell中运行。取决于系统是32位还是64位，要么选择x86 (32位)或x86_64 (64位)安装包。随后你会得到一个文件，名字类似于`Anaconda3-4.1.0-Linux-x86_64.sh`。用bash进行安装：

```
$ bash Anaconda3-4.1.0-Linux-x86_64.sh
```

> 笔记：某些Linux版本在包管理器中有满足需求的Python包，只需用类似apt的工具安装就行。这里讲的用Anaconda安装，适用于不同的Linux安装包，也很容易将包升级到最新版本。

接受许可之后，会向你询问在哪里放置Anaconda的文件。我推荐将文件安装到默认的home目录，例如`/home/$USER/anaconda`。

Anaconda安装包可能会询问你是否将`bin/`目录添加到`$PATH`变量。如果在安装之后有任何问题，你可以修改文件`.bashrc`（或`.zshrc`，如果使用的是zsh shell）为类似以下的内容：

```
export PATH=/home/$USER/anaconda/bin:$PATH
```

做完之后，你可以开启一个新窗口，或再次用`~/.bashrc`执行`.bashrc`。

安装或升级Python包

在你阅读本书的时候，你可能想安装另外的不在Anaconda中的Python包。通常，可以用以下命令安装：

```
conda install package_name
```

如果这个命令不行，也可以用pip包管理工具：

```
pip install package_name
```

你可以用`conda update`命令升级包：

```
conda update package_name
```

pip可以用`--upgrade`升级：

```
pip install --upgrade package_name
```

本书中，你有许多机会尝试这些命令。

> 注意：当你使用conda和pip二者安装包时，千万不要用pip升级conda的包，这样会导致环境发生问题。当使用Anaconda或Miniconda时，最好首先使用conda进行升级。

Python 2 和 Python 3

第一版的Python 3.x出现于2008年。它有一系列的变化，与之前的Python 2.x代码有不兼容的地方。因为从1991年Python出现算起，已经过了17年，Python 3 的出现被视为吸取一些列教训的更优结果。

2012年，因为许多包还没有完全支持Python 3，许多科学和数据分析社区还是在使用Python 2.x。因此，本书第一版使用的是Python 2.7。现在，用户可以在Python 2.x和Python 3.x间自由选择，二者都有良好的支持。

但是，Python 2.x在2020年就会到期（包括重要的安全补丁），因此再用Python 2.7就不是好的选择了。因此，本书使用了Python 3.6，这一广泛使用、支持良好的稳定版本。我们已经称Python 2.x为“遗留版本”，简称Python 3.x为“Python”。我建议你也是如此。

本书基于Python 3.6。你的Python版本也许高于3.6，但是示例代码应该是向前兼容的。一些示例代码可能在Python 2.7上有所不同，或完全不兼容。

集成开发环境（IDEs）和文本编辑器

当被问到我的标准开发环境，我几乎总是回答“IPython加文本编辑器”。我通常在编程时，反复在IPython或Jupyter notebooks中测试和调试每条代码。也可以交互式操作数据，和可视化验证数据操作中某一特殊集合。在shell中使用pandas和NumPy也很容易。

但是，当创建软件时，一些用户可能更想使用特点更为丰富的IDE，而不仅仅是原始的蓄西Emacs或Vim的文本编辑器。以下是一些IDE：

- PyDev（免费），基于Eclipse平台的IDE；
- JetBrains的PyCharm（商业用户需要订阅，开源开发者免费）；
- Visual Studio（Windows用户）的Python Tools；
- Spyder（免费），Anaconda附带的IDE；
- Komodo IDE（商业）。

因为Python的流行，大多数文本编辑器，比如Atom和Sublime Text 3，对Python的支持也非常好。

1.5 社区和会议

除了在网上搜索，各式各样的科学和数据相关的Python邮件列表是非常有帮助的，很容易获得回答。包括：

- pydata：一个Google群组列表，用以回答Python数据分析和pandas的问题；
- pystatsmodels： statsmodels或pandas相关的问题；
- scikit-learn和Python机器学习邮件列表，scikit-learn@python.org；
- numpy-discussion：和NumPy相关的问题；
- scipy-user：SciPy和科学计算的问题；

因为这些邮件列表的URLs可以很容易搜索到，但因为可能发生

变化，所以没有给出。

每年，世界各地会举办许多Python开发者大会。如果你想结识其他有相同兴趣的人，如果可能的话，我建议你去参加一个。许多会议会对无力支付入场费和差旅费的人提供财力帮助。下面是一些会议：

- PyCon和EuroPython：北美和欧洲的两大Python会议；
- SciPy和EuroSciPy：北美和欧洲两大面向科学计算的会议；
- PyData：世界范围内，一些列的地区性会议，专注数据科学和数据分析；
- 国际和地区的PyCon会议（http://pycon.org有完整列表）。

1.6 本书导航

如果之前从未使用过Python，那你可能需要先看看本书的第2章和第3章，我简要介绍了Python的特点，IPython和Jupyter notebooks。这些知识是为本书后面的内容做铺垫。如果你已经掌握Python，可以选择跳过。

接下来，简单地介绍了NumPy的关键特性，附录A中是更高级的NumPy功能。然后，我介绍了pandas，本书剩余的内容全部是使用pandas、NumPy和matplotlib处理数据分析的问题。我已经尽量让全书的结构循序渐进，但偶尔会有章节之间的交叉，有时用到的概念还没有介绍过。

尽管读者各自的工作任务不同，大体可以分为几类：

- 与外部世界交互
  阅读编写多种文件格式和数据商店；
- 数据准备
  清洗、修改、结合、标准化、重塑、切片、切割、转换数据，以进行分析；
- 转换数据
  对旧的数据集进行数学和统计操作，生成新的数据集（例如，通过各组变量聚类成大的表）；

- 建模和计算
  将数据绑定统计模型、机器学习算法、或其他计算工具；
- 展示
  创建交互式和静态的图表可视化和文本总结。

代码示例

本书大部分代码示例的输入形式和输出结果都会按照其在IPython shell或Jupyter notebooks中执行时的样子进行排版：

```
In [5]: CODE EXAMPLE
Out[5]: OUTPUT
```

但你看到类似的示例代码，就是让你在`in`的部分输入代码，按Enter键执行（Jupyter中是按Shift-Enter）。然后就可以在`out`看到输出。

示例数据

各章的示例数据都存放在GitHub上：http://github.com/pydata/pydata-book。下载这些数据的方法有二：使用git版本控制命令行程序；直接从网站上下载该GitHub库的zip文件。如果遇到了问题，可以到我的个人主页，http://wesmckinney.com/，获取最新的指导。

为了让所有示例都能重现，我已经尽我所能使其包含所有必需的东西，但仍然可能会有一些错误或遗漏。如果出现这种情况的话，请给我发邮件：wesmckinn@gmail.com。报告本书错误的最好方法是O’Reilly的errata页面，http://www.bit.ly/pyDataAnalysis_errata。

引入惯例

Python社区已经广泛采取了一些常用模块的命名惯例：

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import pandas as pd
import seaborn as sns
import statsmodels as sm
```

也就是说，当你看到np.arange时，就应该想到它引用的是NumPy中的arange函数。这样做的原因是：在Python软件开发过程中，不建议直接引入类似NumPy这种大型库的全部内容（from numpy import *）。

行话

由于你可能不太熟悉书中使用的一些有关编程和数据科学方面的常用术语，所以我在这里先给出其简单定义：

数据规整（Munge/Munging/Wrangling）
指的是将非结构化和（或）散乱数据处理为结构化或整洁形式的整个过程。这几个词已经悄悄成为当今数据黑客们的行话了。Munge这个词跟Lunge押韵。

伪码（Pseudocode）
算法或过程的“代码式”描述，而这些代码本身并不是实际有效的源代码。

语法糖（Syntactic sugar）
这是一种编程语法，它并不会带来新的特性，但却能使代码更易读、更易写。

第2章 Python语法基础，IPython和Jupyter Notebooks

当我在2011年和2012年写作本书的第一版时，可用的学习Python数据分析的资源很少。这部分上是一个鸡和蛋的问题：我们现在使用的库，比如pandas、scikit-learn和statsmodels，那时相对来说并不成熟。2017年，数据科学、数据分析和机器学习的资源已经很多，原来通用的科学计算拓展到了计算机科学家、物理学家和其它研究领域的工作人员。学习Python和成为软件工程师的优秀书籍也有了。

因为这本书是专注于Python数据处理的，对于一些Python的数据结构和库的特性难免不足。因此，本章和第3章的内容只够你能学习本书后面的内容。

在我来看，没有必要为了数据分析而去精通Python。我鼓励你使用IPython shell和Jupyter试验示例代码，并学习不同类型、函数和方法的文档。虽然我已尽力让本书内容循序渐进，但读者偶尔仍会碰到没有之前介绍过的内容。

本书大部分内容关注的是基于表格的分析和处理大规模数据集的数据准备工具。为了使用这些工具，必须首先将混乱的数据规整为整洁的表格（或结构化）形式。幸好，Python是一个理想的语言，可以快速整理数据。使用Python越熟练，越容易准备新的数据集以进行分析。

本书中使用的工具最好在IPython和Jupyter中亲自尝试。当你学会了如何启用Ipython和Jupyter，我建议你跟随示例代码进行练习。与任何键盘驱动的操作环境一样，记住常见的命令也是学习曲线的一部分。

> 笔记：本章没有介绍Python的某些概念，如类和面向对象编程，你可能会发现它们在Python数据分析中很有用。
> 为了加强Python知识，我建议你学习官方Python教程，https://docs.python.org/3/，或是通用的Python教程书

> 籍，比如：
>
> - Python Cookbook，第3版，David Beazley和Brian K. Jones著（O'Reilly）
> - 流畅的Python，Luciano Ramalho著 (O'Reilly)
> - 高效的Python，Brett Slatkin著 (Pearson)

2.1 Python解释器

Python是解释性语言。Python解释器同一时间只能运行一个程序的一条语句。标准的交互Python解释器可以在命令行中通过键入`python`命令打开：

```
$ python
Python 3.6.0 | packaged by conda-forge | (default, Ja
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for 
>>> a = 5
>>> print(a)
5
```

`>>>`提示输入代码。要退出Python解释器返回终端，可以输入`exit()`或按Ctrl-D。

运行Python程序只需调用Python的同时，使用一个`.py`文件作为它的第一个参数。假设创建了一个`hello_world.py`文件，它的内容是：

```
print('Hello world')
```

你可以用下面的命令运行它（`hello_world.py`文件必须位于终端的工作目录）：

```
$ python hello_world.py
Hello world
```

一些Python程序员总是这样执行Python代码的，从事数据分析和科学计算的人却会使用IPython，一个强化的Python解释器，或Jupyter notebooks，一个网页代码笔记本，它原先是IPython的一个子项目。在本章中，我介绍了如何使用IPython和Jupyter，在附录A中有更深入的介绍。当你使用`%run`命令，IPython会同样执行指定文件中的代码，结束之后，还可以与结果交互：

```
$ ipython
Python 3.6.0 | packaged by conda-forge | (default, Ja
Type "copyright", "credits" or "license" for more inf

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's f
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' f

In [1]: %run hello_world.py
Hello world

In [2]:
```

IPython默认采用序号的格式`In [2]:`，与标准的`>>>`提示符不同。

2.2 IPython基础

在本节中，我们会教你打开运行IPython shell和jupyter notebook，并介绍一些基本概念。

运行IPython Shell

你可以用`ipython`在命令行打开IPython Shell，就像打开普通的Python解释器：

```
$ ipython
```

```
Python 3.6.0 | packaged by conda-forge | (default, Ja
Type "copyright", "credits" or "license" for more inf

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's f
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' f

In [1]: a = 5
In [2]: a
Out[2]: 5
```

你可以通过输入代码并按Return（或Enter），运行任意Python语句。当你只输入一个变量，它会显示代表的对象：

```
In [5]: import numpy as np

In [6]: data = {i : np.random.randn() for i in range(

In [7]: data
Out[7]:
{0: -0.20470765948471295,
 1: 0.47894333805754824,
 2: -0.5194387150567381,
 3: -0.55573030434749,
 4: 1.9657805725027142,
 5: 1.3934058329729904,
6: 0.09290787674371767}
```

前两行是Python代码语句；第二条语句创建一个名为`data`的变量，它引用一个新创建的Python字典。最后一行打印`data`的值。

许多Python对象被格式化为更易读的形式，或称

作`pretty-printed`，它与普通的`print`不同。如果在标准Python解释器中打印上述`data`变量，则可读性要降低：

```
>>> from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> print(data)
{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1
3: 0.1545517573074576, 4: -0.24058577449429575, 5: -1
6: 0.3308507317325902}
```

IPython还支持执行任意代码块（通过一个华丽的复制-粘贴方法）和整段Python脚本的功能。你也可以使用Jupyter notebook运行大代码块，接下来就会看到。

运行Jupyter Notebook

Jupyter项目的重要组件之久就是notebook，它是一个代码、文本（有标记或无标记）、数据可视化或其它输出的交互式文档。Jupyter Notebook需要与内核互动，内核是Jupyter与其它编程语言的交互编程协议。Python的Jupyter内核是使用IPython。要启动Jupyter，在命令行中输入`jupyter notebook`：

```
$ jupyter notebook
[I 15:20:52.739 NotebookApp] Serving notebooks from lo
/home/wesm/code/pydata-book
[I 15:20:52.739 NotebookApp] 0 active kernels
[I 15:20:52.739 NotebookApp] The Jupyter Notebook is
http://localhost:8888/
[I 15:20:52.740 NotebookApp] Use Control-C to stop th
all kernels (twice to skip confirmation).
Created new window in existing browser session.
```

在多数平台上，Jupyter会自动打开默认的浏览器（除非指定了`--no-browser`）。或者，可以在启动notebook之后，手动打开网页`http://localhost:8888/`。图2-1展示了Google Chrome

中的notebook。

> 笔记：许多人使用Jupyter作为本地的计算环境，但它也可以部署到服务器上远程访问。这里不做介绍，如果需要的话，鼓励读者自行到网上学习。



图2-1 Jupyter notebook启动页面

要新建一个notebook，点击按钮New，选择“Python3”或“conda[默认项]”。如果是第一次，点击空格，输入一行Python代码。然后按Shift-Enter执行。

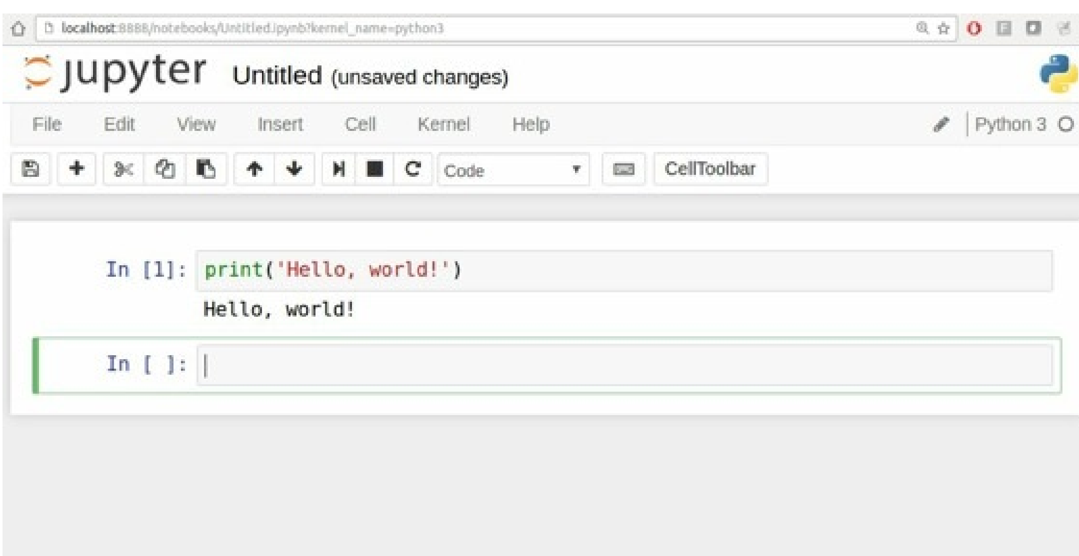


图2-2 Jupyter新notebook页面

当保存notebook时（File目录下的Save and Checkpoint），会创建一个后缀名为`.ipynb`的文件。这是一个自包含文件格式，包含当前笔记本中的所有内容（包括所有已评估的代码输出）。可以被其它Jupyter用户加载和编辑。要加载存在的notebook，把它放到启动notebook进程的相同目录内。你可以用本书的示例代码练习，见图2-3。

虽然Jupyter notebook和IPython shell使用起来不同，本章中几乎所有的命令和工具都可以通用。

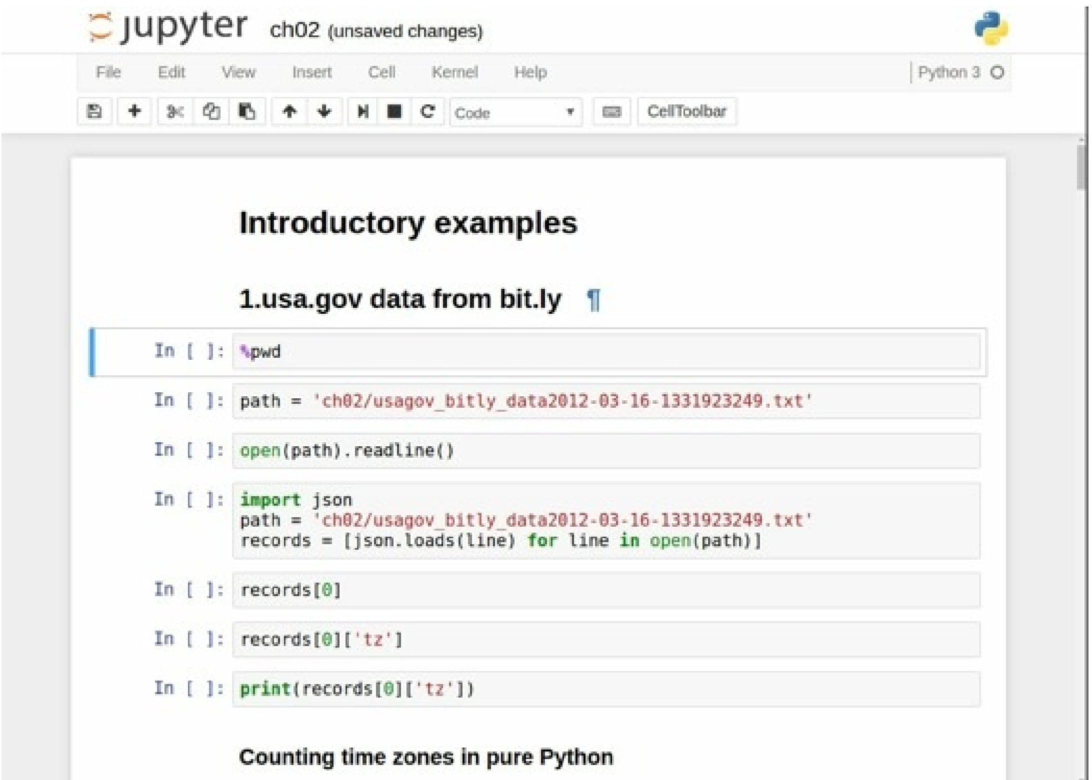


图2-3 Jupyter查看一个存在的notebook的页面

Tab补全

从外观上，IPython shell和标准的Python解释器只是看起来不同。IPython shell的进步之一是其它IDE和交互计算分析环境都有的tab补全功能。在shell中输入表达式，按下Tab，会搜索已输入变量（对象、函数等等）的命名空间：

```
In [1]: an_apple = 27
```

```
In [2]: an_example = 42

In [3]: an<Tab>
an_apple    and         an_example  any
```

在这个例子中，IPython呈现除了之前两个定义的变量和Python的关键字和内建的函数`any`。当然，你也可以补全任何对象的方法和属性：

```
In [3]: b = [1, 2, 3]

In [4]: b.<Tab>
b.append  b.count   b.insert  b.reverse
b.clear   b.extend  b.pop     b.sort
b.copy    b.index   b.remove
```

同样也适用于模块：

```
In [1]: import datetime

In [2]: datetime.<Tab>
datetime.date          datetime.MAXYEAR       datetim
datetime.datetime      datetime.MINYEAR       datetim
datetime.datetime_CAPI datetime.time          datetim
```

在Jupyter notebook和新版的IPython（5.0及以上），自动补全功能是下拉框的形式。

> 笔记：注意，默认情况下，IPython会隐藏下划线开头的方法和属性，比如魔术方法和内部的“私有”方法和属性，以避免混乱的显示（和让新手迷惑！）这些也可以tab补全，但是你必须首先键入一个下划线才能看到它们。如果你喜欢总是在tab补全中看到这样的方法，你可以IPython配置中进行设置。可以在IPython文档中查找方法。

除了补全命名、对象和模块属性，Tab还可以补全其它的。当输入看似文件路径时（即使是Python字符串），按下Tab也可以补全电脑上对应的文件信息：

```
In [7]: datasets/movielens/<Tab>
datasets/movielens/movies.dat    datasets/movielens/R
datasets/movielens/ratings.dat   datasets/movielens/u

In [7]: path = 'datasets/movielens/<Tab>
datasets/movielens/movies.dat    datasets/movielens/R
datasets/movielens/ratings.dat   datasets/movielens/u
```

结合`%run`，tab补全可以节省许多键盘操作。

另外，tab补全可以补全函数的关键词参数（包括等于号=）。见图2-4。

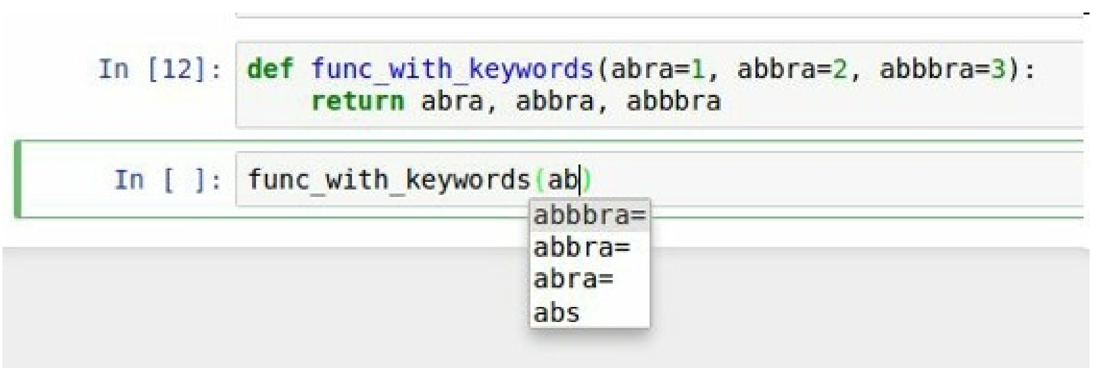


图2-4 Jupyter notebook中自动补全函数关键词

我们来仔细看看函数。

自省

在变量前后使用问号？，可以显示对象的信息：

```
In [8]: b = [1, 2, 3]

In [9]: b?
Type:        list
```

```
String Form:[1, 2, 3]
Length:     3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable'

In [10]: print?
Docstring:
print(value, ..., sep=' ', end='\n', file=sys.stdout,

Prints the values to a stream, or to sys.stdout by de
Optional keyword arguments:
file:  a file-like object (stream); defaults to the c
sep:   string inserted between values, default a spac
end:   string appended after the last value, default 
flush: whether to forcibly flush the stream.
Type:      builtin_function_or_method
```

这可以作为对象的自省。如果对象是一个函数或实例方法，定义过的文档字符串，也会显示出信息。假设我们写了一个如下的函数：

```
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
```

然后使用?符号，就可以显示如下的文档字符串：

```
In [11]: add_numbers?
Signature: add_numbers(a, b)
```

```
Docstring:
Add two numbers together

Returns
-------
the_sum : type of arguments
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

使用??会显示函数的源码：

```
In [12]: add_numbers??
Signature: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
File:      <ipython-input-9-6a548a216e27>
Type:      function
```

?还有一个用途，就是像Unix或Windows命令行一样搜索IPython的命名空间。字符与通配符结合可以匹配所有的名字。例如，我们可以获得所有包含load的顶级NumPy命名空间：

```
In [13]: np.*load*?
np.__loader__
np.load
np.loads
np.loadtxt
np.pkgload
```

%run命令

你可以用`%run`命令运行所有的Python程序。假设有一个文件`ipython_script_test.py`：

```
def f(x, y, z):
    return (x + y) / z

a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

可以如下运行：

```
In [14]: %run ipython_script_test.py
```

这段脚本运行在空的命名空间（没有import和其它定义的变量），因此结果和普通的运行方式`python script.py`相同。文件中所有定义的变量（import、函数和全局变量，除非抛出异常），都可以在IPython shell中随后访问：

```
In [15]: c
Out [15]: 7.5

In [16]: result
Out[16]: 1.4666666666666666
```

如果一个Python脚本需要命令行参数（在`sys.argv`中查找），可以在文件路径之后传递，就像在命令行上运行一样。

> 笔记：如果想让一个脚本访问IPython已经定义过的变量，可以使用`%run -i`。

在Jupyter notebook中，你也可以使用`%load`，它将脚本导入到一个代码格中：

```
>>> %load ipython_script_test.py

    def f(x, y, z):
        return (x + y) / z
    a = 5
    b = 6
    c = 7.5

    result = f(a, b, c)
```

中断运行的代码

代码运行时按Ctrl-C，无论是%run或长时间运行命令，都会导致`KeyboardInterrupt`。这会导致几乎左右Python程序立即停止，除非一些特殊情况。

> 警告：当Python代码调用了一些编译的扩展模块，按Ctrl-C不一定将执行的程序立即停止。在这种情况下，你必须等待，直到控制返回Python解释器，或者在更糟糕的情况下强制终止Python进程。

从剪贴板执行程序

如果使用Jupyter notebook，你可以将代码复制粘贴到任意代码格执行。在IPython shell中也可以从剪贴板执行。假设在其它应用中复制了如下代码：

```
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
```

最简单的方法是使用`%paste`和`%cpaste`函数。`%paste`可以直接运行剪贴板中的代码：

```
In [17]: %paste
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
## -- End pasted text --
```

`%cpaste`功能类似，但会给出一条提示：

```
In [18]: %cpaste
Pasting code; enter '--' alone on the line to stop or
:x = 5
:y = 7
:if x > 5:
:    x += 1
:
:    y = 8
:--
```

使用`%cpaste`，你可以粘贴任意多的代码再运行。你可能想在运行前，先看看代码。如果粘贴了错误的代码，可以用Ctrl-C中断。

键盘快捷键

IPython有许多键盘快捷键进行导航提示（类似Emacs文本编辑器或UNIX bash Shell）和交互shell的历史命令。表2-1总结了常见的快捷键。图2-5展示了一部分，如移动光标。

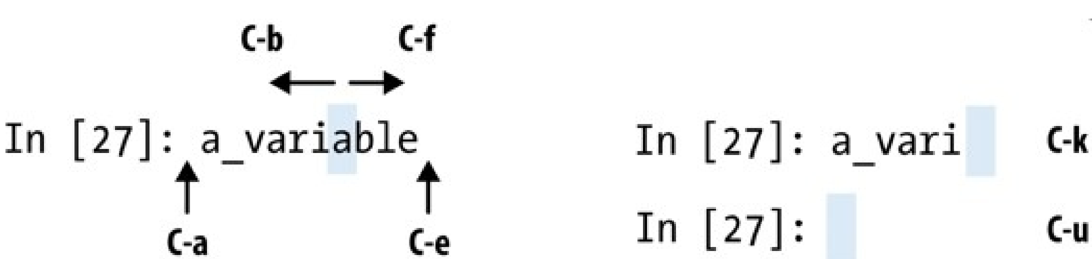


图2-5 IPython shell中一些快捷键的说明

快捷键	说明
Ctrl-P 或↑箭头	用当前输入的文本搜索之前的命令
Ctrl-N 或↓箭头	用当前输入的文本搜索之后的命令
Ctrl-R	Readline 方式翻转历史搜索（部分匹配）
Ctrl-Shift-V	从剪贴板粘贴文本
Ctrl-C	中断运行的代码
Ctrl-A	将光标移动到一行的开头
Ctrl-E	将光标移动到一行的末尾
Ctrl-K	删除光标到行尾的文本
Ctrl-U	删除当前行的所有文本
Ctrl-F	光标向后移动一个字符
Ctrl-B	光标向前移动一个字符
Ctrl-L	清空屏幕

表2-1 IPython的标准快捷键

Jupyter notebooks有另外一套庞大的快捷键。因为它的快捷键比IPython的变化快，建议你参阅Jupyter notebook的帮助文档。

魔术命令

IPython中特殊的命令（Python中没有）被称作“魔术”命令。这些命令可以使普通任务更便捷，更容易控制IPython系统。魔术命令是在指令前添加百分号%前缀。例如，可以用`%timeit`（这个命令后面会详谈）测量任何Python语句，例如矩阵乘法，的执行时间：

```
In [20]: a = np.random.randn(100, 100)

In [20]: %timeit np.dot(a, a)
10000 loops, best of 3: 20.9 µs per loop
```

魔术命令可以被看做IPython中运行的命令行。许多魔术命令有“命令行”选项，可以通过？查看：

```
In [21]: %debug?
Docstring:
::

  %debug [--breakpoint FILE:LINE] [statement [stateme

Activate the interactive debugger.

This magic command support two ways of activating deb
One is to activate debugger before executing code.  T
can set a break point, to step through the code from 
You can use this mode by giving statements to execute
a breakpoint.

The other one is to activate debugger in post-mortem 
activate this mode simply running %debug without any 
If an exception has just occurred, this lets you insp
frames interactively.  Note that this will always wor
traceback that occurred, so you must call this quickl
exception that you wish to inspect has fired, because
occurs, it clobbers the previous one.

If you want IPython to automatically do this on every
the %pdb magic for more details.

positional arguments:
  statement             Code to run in debugger. You 
                        magic mode.

optional arguments:
  --breakpoint <FILE:LINE>, -b <FILE:LINE>
                        Set break point at LINE in FI
```

魔术函数默认可以不用百分号，只要没有变量和函数名相同。这个特点被称为“自动魔术”，可以用`%automagic`打开或关闭。

一些魔术函数与Python函数很像，它的结果可以赋值给一个变量：

```
In [22]: %pwd
Out[22]: '/home/wesm/code/pydata-book

In [23]: foo = %pwd

In [24]: foo
Out[24]: '/home/wesm/code/pydata-book'
```

IPython的文档可以在shell中打开，我建议你用`%quickref`或`%magic`学习下所有特殊命令。表2-2列出了一些可以提高生产率的交互计算和Python开发的IPython指令。

命令	说明
%quickref	显示 IPython 的快速参考。
%magic	显示所有魔术命令的详细文档。
%debug	在出现异常的语句进入调试模式。
%hist	打印命令的输入（可以选择输出）历史。
%pdb	出现异常时自动进入调试。
%paste	执行剪贴板中的代码。
%cpaste	开启特别提示，手动粘贴待执行代码。
%reset	删除所有命名空间中的变量和名字。
%page OBJECT	美化打印对象，分页显示。
%run script.py	运行代码。
%prun statement	用 CProfile 运行代码，并报告分析器输出。
%time statement	报告单条语句的执行时间。
%timeit statement	多次运行一条语句，计算平均执行时间。适合执行时间短的代码。
%who, %who_ls, %whos	显示命名空间中的变量，三者显示的信息级别不同。
%xdel variable	删除一个变量，并清空任何对它的引用。

表2-2 一些常用的IPython魔术命令

集成Matplotlib

IPython在分析计算领域能够流行的原因之一是它非常好的集成了数据可视化和其它用户界面库，比如matplotlib。不用担心以前没用过matplotlib，本书后面会详细介绍。`%matplotlib`魔术函

数配置了IPython shell和Jupyter notebook中的matplotlib。这点很重要，其它创建的图不会出现（notebook）或获取session的控制，直到结束（shell）。

在IPython shell中，运行`%matplotlib`可以进行设置，可以创建多个绘图窗口，而不会干扰控制台session：

```
In [26]: %matplotlib
Using matplotlib backend: Qt4Agg
```

在JUpyter中，命令有所不同（图2-6）：

```
In [26]: %matplotlib inline
```

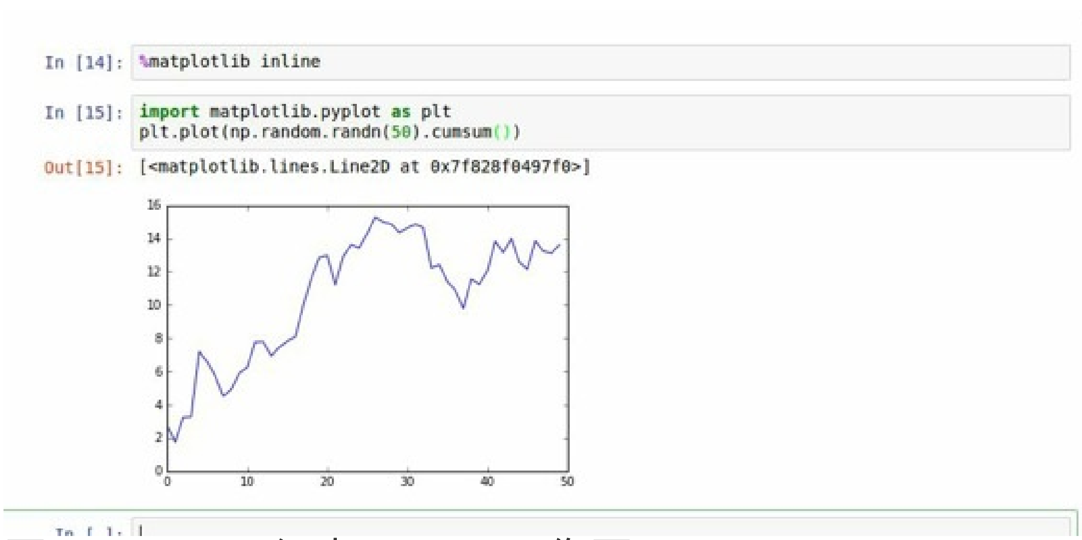


图2-6 Jupyter行内matplotlib作图

2.3 Python语法基础

在本节中，我将概述基本的Python概念和语言机制。在下一章，我将详细介绍Python的数据结构、函数和其它内建工具。

语言的语义

Python的语言设计强调的是可读性、简洁和清晰。有些人称Python为“可执行的伪代码”。

使用缩进，而不是括号

Python使用空白字符（tab和空格）来组织代码，而不是像其它语言，比如R、C++、JAVA和Perl那样使用括号。看一个排序算法的`for`循环：

```
for x in array:
    if x < pivot:
        less.append(x)
    else:
        greater.append(x)
```

冒号标志着缩进代码块的开始，冒号之后的所有代码的缩进量必须相同，直到代码块结束。不管是否喜欢这种形式，使用空白符是Python程序员开发的一部分，在我看来，这可以让python的代码可读性大大优于其它语言。虽然期初看起来很奇怪，经过一段时间，你就能适应了。

> 笔记：我强烈建议你使用四个空格作为默认的缩进，可以使用tab代替四个空格。许多文本编辑器的设置是使用制表位替代空格。某些人使用tabs或不同数目的空格数，常见的是使用两个空格。大多数情况下，四个空格是大多数人采用的方法，因此建议你也这样做。

你应该已经看到，Python的语句不需要用分号结尾。但是，分号却可以用来给同在一行的语句切分：

```
a = 5; b = 6; c = 7
```

Python不建议将多条语句放到一行，这会降低代码的可读性。

万物皆对象

Python语言的一个重要特性就是它的对象模型的一致性。每个数字、字符串、数据结构、函数、类、模块等等，都是在Python解释器的自有“盒子”内，它被认为是Python对象。每个对象都有类

型（例如，字符串或函数）和内部数据。在实际中，这可以让语言非常灵活，因为函数也可以被当做对象使用。

注释

任何前面带有井号#的文本都会被Python解释器忽略。这通常被用来添加注释。有时，你会想排除一段代码，但并不删除。简便的方法就是将其注释掉：

```
results = []
for line in file_handle:
    # keep the empty lines for now
    # if len(line) == 0:
    #   continue
    results.append(line.replace('foo', 'bar'))
```

也可以在执行过的代码后面添加注释。一些人习惯在代码之前添加注释，前者这种方法有时也是有用的：

```
print("Reached this line")  # Simple status report
```

函数和对象方法调用

你可以用圆括号调用函数，传递零个或几个参数，或者将返回值给一个变量：

```
result = f(x, y, z)
g()
```

几乎Python中的每个对象都有附加的函数，称作方法，可以用来访问对象的内容。可以用下面的语句调用：

```
obj.some_method(x, y, z)
```

函数可以使用位置和关键词参数：

```
result = f(a, b, c, d=5, e='foo')
```

后面会有更多介绍。

变量和参数传递

当在Python中创建变量（或名字），你就在等号右边创建了一个对这个变量的引用。考虑一个整数列表：

```
In [8]: a = [1, 2, 3]
```

假设将a赋值给一个新变量b：

```
In [9]: b = a
```

在有些方法中，这个赋值会将数据[1, 2, 3]也复制。在Python中，a和b实际上是同一个对象，即原有列表[1, 2, 3]（见图2-7）。你可以在a中添加一个元素，然后检查b：

```
In [10]: a.append(4)

In [11]: b
Out[11]: [1, 2, 3, 4]
```

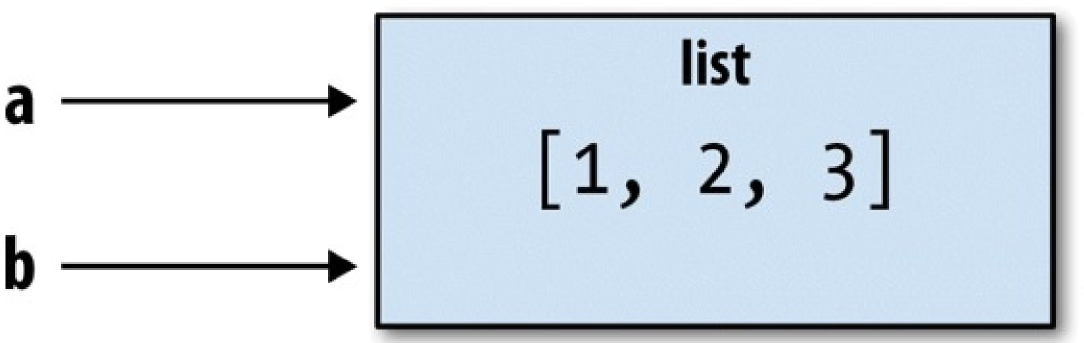


图2-7 对同一对象的双重引用

理解Python的引用的含义，数据是何时、如何、为何复制的，是非常重要的。尤其是当你用Python处理大的数据集时。

> 笔记：赋值也被称作绑定，我们是把一个名字绑定给一个对象。变量名有时可能被称为绑定变量。

当你将对象作为参数传递给函数时，新的局域变量创建了对原始对象的引用，而不是复制。如果在函数里绑定一个新对象到一个变量，这个变动不会反映到上一层。因此可以改变可变参数的内容。假设有以下函数：

```
def append_element(some_list, element):
    some_list.append(element)
```

然后有：

```
In [27]: data = [1, 2, 3]

In [28]: append_element(data, 4)

In [29]: data
Out[29]: [1, 2, 3, 4]
```

动态引用，强类型

与许多编译语言（如JAVA和C++）对比，Python中的对象引用不包含附属的类型。下面的代码是没有问题的：

```
In [12]: a = 5

In [13]: type(a)
Out[13]: int

In [14]: a = 'foo'

In [15]: type(a)
Out[15]: str
```

变量是在特殊命名空间中的对象的名字，类型信息保存在对象自身中。一些人可能会说Python不是“类型化语言”。这是不正确的，看下面的例子：

```
In [16]: '5' + 5
---------------------------------------------------------
TypeError                                 Traceback (r
<ipython-input-16-f9dbf5f0b234> in <module>()
----> 1 '5' + 5
TypeError: must be str, not int
```

在某些语言中，例如Visual Basic，字符串‘5’可能被默许转换（或投射）为整数，因此会产生10。但在其它语言中，例如JavaScript，整数5个能被投射成字符串，结果是联结字符串‘55’。在这个方面，Python被认为是强类型化语言，意味着每个对象都有明确的类型（或类），默许转换只会发生在特定的情况下，例如：

```
In [17]: a = 4.5

In [18]: b = 2

# String formatting, to be visited later
In [19]: print('a is {0}, b is {1}'.format(type(a), ty
a is <class 'float'>, b is <class 'int'>

In [20]: a / b
Out[20]: 2.25
```

知道对象的类型很重要，最好能让函数可以处理多种类型的输入。你可以用`isinstance`函数检查对象是某个类型的实例：

```
In [21]: a = 5

In [22]: isinstance(a, int)
Out[22]: True
```

`isinstance`可以用类型元组，检查对象的类型是否在元组中：

```
In [23]: a = 5; b = 4.5

In [24]: isinstance(a, (int, float))
Out[24]: True

In [25]: isinstance(b, (int, float))
Out[25]: True
```

属性和方法

Python的对象通常都有属性（其它存储在对象内部的Python对象）和方法（对象的附属函数可以访问对象的内部数据）。可以用`obj.attribute_name`访问属性和方法：

```
In [1]: a = 'foo'

In [2]: a.<Press Tab>
a.capitalize  a.format      a.isupper     a.rindex
a.center      a.index       a.join        a.rjust
a.count       a.isalnum     a.ljust       a.rpartitio
a.decode      a.isalpha     a.lower       a.rsplit
a.encode      a.isdigit     a.lstrip      a.rstrip
a.endswith    a.islower     a.partition   a.split
a.expandtabs  a.isspace     a.replace     a.splitline
a.find        a.istitle     a.rfind       a.startswit
```

也可以用`getattr`函数，通过名字访问属性和方法：

```
In [27]: getattr(a, 'split')
Out[27]: <function str.split>
```

在其它语言中，访问对象的名字通常称作“反射”。本书不会大量使用`getattr`函数和相关的`hasattr`和`setattr`函数，使用这些函数可以高效编写原生的、可重复使用的代码。

鸭子类型

经常地，你可能不关心对象的类型，只关心对象是否有某些方法或用途。这通常被称为“鸭子类型”，来自“走起来像鸭子、叫起来像鸭子，那么它就是鸭子”的说法。例如，你可以通过验证一个对象是否遵循迭代协议，判断它是可迭代的。对于许多对象，这意味着它有一个`__iter__`魔术方法，其它更好的判断方法是使用`iter`函数：

```
def isiterable(obj):
    try:
        iter(obj)
        return True
    except TypeError: # not iterable
        return False
```

这个函数会返回字符串以及大多数Python集合类型为`True`：

```
In [29]: isiterable('a string')
Out[29]: True

In [30]: isiterable([1, 2, 3])
Out[30]: True

In [31]: isiterable(5)
Out[31]: False
```

我总是用这个功能编写可以接受多种输入类型的函数。常见的例子是编写一个函数可以接受任意类型的序列（list、tuple、ndarray）或是迭代器。你可先检验对象是否是列表（或是NUmPy数组），如果不是的话，将其转变成列表：

```
if not isinstance(x, list) and isiterable(x):
    x = list(x)
```

引入

在Python中，模块就是一个有`.py`扩展名、包含Python代码的文

件。假设有以下模块：

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

如果想从同目录下的另一个文件访问`some_module.py`中定义的变量和函数，可以：

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

或者：

```
from some_module import f, g, PI
result = g(5, PI)
```

使用`as`关键词，你可以给引入起不同的变量名：

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

二元运算符和比较运算符

大多数二元数学运算和比较都不难想到：

```
In [32]: 5 - 7
Out[32]: -2
```

```
In [33]: 12 + 21.5
Out[33]: 33.5

In [34]: 5 <= 2
Out[34]: False
```

表2-3列出了所有的二元运算符。

要判断两个引用是否指向同一个对象，可以使用`is`方法。`is not`可以判断两个对象是不同的：

```
In [35]: a = [1, 2, 3]

In [36]: b = a

In [37]: c = list(a)

In [38]: a is b
Out[38]: True

In [39]: a is not c
Out[39]: True
```

因为`list`总是创建一个新的Python列表（即复制），我们可以断定c是不同于a的。使用`is`比较与`==`运算符不同，如下：

```
In [40]: a == c
Out[40]: True
```

`is`和`is not`常用来判断一个变量是否为`None`，因为只有一个`None`的实例：

```
In [41]: a = None

In [42]: a is None
Out[42]: True
```

运算	说明
a + b	a 加 b
a - b	a 减 b
a * b	a 乘以 b
a / b	a 除以 b
a // b	a 除以 b，结果只取整数部分
a ** b	a 的 b 次幂
a & b	a 或 b 都为 True，则为 True；对于整数，取逐位 AND
a | b	a 或 b 有一个为 True，则为 True；对于整数，取逐位 OR
a ^ b	对于布尔，a 或 b 有一个为 True，则为 True，二者都为 True，为 False；对于整数，取逐位 EXCLUSIVE-OR
a == b	a 等于 b，则为 True
a != b	a 不等于 b，则为 True
a < b, a <= b	a 小于（或小于等于）b，则为 True
a > b, a >= b	a 大于（或大于等于）b，则为 True
a is b	a 和 b 引用同一个 Python 对象，则为 True
a is not b	a 和 b 引用不同的 Python 对象，则为 True

表2-3 二元运算符

可变与不可变对象

Python中的大多数对象，比如列表、字典、NumPy数组，和用户定义的类型（类），都是可变的。意味着这些对象或包含的值可以被修改：

```
In [43]: a_list = ['foo', 2, [4, 5]]

In [44]: a_list[2] = (3, 4)

In [45]: a_list
Out[45]: ['foo', 2, (3, 4)]
```

其它的，例如字符串和元组，是不可变的：

```
In [46]: a_tuple = (3, 5, (4, 5))

In [47]: a_tuple[1] = 'four'
---------------------------------------------------------
TypeError                                 Traceback (
```

```
<ipython-input-47-b7966a9ae0f1> in <module>()
----> 1 a_tuple[1] = 'four'
TypeError: 'tuple' object does not support item assign
```

记住，可以修改一个对象并不意味就要修改它。这被称为副作用。例如，当写一个函数，任何副作用都要在文档或注释中写明。如果可能的话，我推荐避免副作用，采用不可变的方式，即使要用到可变对象。

标量类型

Python的标准库中有一些内建的类型，用以处理数值数据、字符串、布尔值，和日期时间。这些单值类型被称为标量类型，本书中称其为标量。表2-4列出了主要的标量。日期和时间处理会另外讨论，因为它们是标准库的`datetime`模块提供的。

类型	说明
None	Python 的空值（只存在一个 None 对象的实例）
str	字符串类型，存有 Unicode（UTF-8 编码）字符串
bytes	原生 ASCII 字节（或 Unicode 编码为字节）
float	双精度（64 位）浮点数（注意没有 double 类型）
bool	True 或 False 值
int	任意精度整数

表2-4 Python的标量

数值类型

Python的主要数值类型是`int`和`float`。`int`可以存储任意大的数：

```
In [48]: ival = 17239871

In [49]: ival ** 6
Out[49]: 26254519291092456596965462913230729701102721
```

浮点数使用Python的`float`类型。每个数都是双精度（64位）的

值。也可以用科学计数法表示：

```
In [50]: fval = 7.243

In [51]: fval2 = 6.78e-5
```

不能得到整数的除法会得到浮点数：

```
In [52]: 3 / 2
Out[52]: 1.5
```

要获得C-风格的整除（去掉小数部分），可以使用底除运算符//：

```
In [53]: 3 // 2
Out[53]: 1
```

字符串

许多人是因为Python强大而灵活的字符串处理而使用Python的。你可以用单引号或双引号来写字符串：

```
a = 'one way of writing a string'
b = "another way"
```

对于有换行符的字符串，可以使用三引号，'''或"""都行：

```
c = """
This is a longer string that
spans multiple lines
"""
```

字符串`c`实际包含四行文本，"""后面和lines后面的换行符。可以用`count`方法计算`c`中的新的行：

```
In [55]: c.count('\n')
Out[55]: 3
```

Python的字符串是不可变的，不能修改字符串：

```
In [56]: a = 'this is a string'

In [57]: a[10] = 'f'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment

In [58]: b = a.replace('string', 'longer string')

In [59]: b
Out[59]: 'this is a longer string'
```

经过以上的操作，变量`a`并没有被修改：

```
In [60]: a
Out[60]: 'this is a string'
```

许多Python对象使用`str`函数可以被转化为字符串：

```
In [61]: a = 5.6

In [62]: s = str(a)

In [63]: print(s)
5.6
```

字符串是一个序列的Unicode字符，因此可以像其它序列，比如列表和元组（下一章会详细介绍两者）一样处理：

```
In [64]: s = 'python'

In [65]: list(s)
```

```
Out[65]: ['p', 'y', 't', 'h', 'o', 'n']

In [66]: s[:3]
Out[66]: 'pyt'
```

语法`s[:3]`被称作切片，适用于许多Python序列。后面会更详细的介绍，本书中用到很多切片。

反斜杠是转义字符，意思是它备用来表示特殊字符，比如换行符\n或Unicode字符。要写一个包含反斜杠的字符串，需要进行转义：

```
In [67]: s = '12\\34'

In [68]: print(s)
12\34
```

如果字符串中包含许多反斜杠，但没有特殊字符，这样做就很麻烦。幸好，可以在字符串前面加一个r，表明字符就是它自身：

```
In [69]: s = r'this\has\no\special\characters'

In [70]: s
Out[70]: 'this\\has\\no\\special\\characters'
```

r表示raw。

将两个字符串合并，会产生一个新的字符串：

```
In [71]: a = 'this is the first half '

In [72]: b = 'and this is the second half'

In [73]: a + b
Out[73]: 'this is the first half and this is the secon
```

字符串的模板化或格式化，是另一个重要的主题。Python 3拓展了此类的方法，这里只介绍一些。字符串对象有`format`方法，可以替换格式化的参数为字符串，产生一个新的字符串：

```
In [74]: template = '{0:.2f} {1:s} are worth US${2:d}
```

在这个字符串中，

- `{0:.2f}`表示格式化第一个参数为带有两位小数的浮点数。
- `{1:s}`表示格式化第二个参数为字符串。
- `{2:d}`表示格式化第三个参数为一个整数。

要替换参数为这些格式化的参数，我们传递`format`方法一个序列：

```
In [75]: template.format(4.5560, 'Argentine Pesos', 1
Out[75]: '4.56 Argentine Pesos are worth US$1'
```

字符串格式化是一个很深的主题，有多种方法和大量的选项，可以控制字符串中的值是如何格式化的。推荐参阅Python官方文档。

这里概括介绍字符串处理，第8章的数据分析会详细介绍。

字节和Unicode

在Python 3及以上版本中，Unicode是一级的字符串类型，这样可以更一致的处理ASCII和Non-ASCII文本。在老的Python版本中，字符串都是字节，不使用Unicode编码。假如知道字符编码，可以将其转化为Unicode。看一个例子：

```
In [76]: val = "español"

In [77]: val
Out[77]: 'español'
```

可以用`encode`将这个Unicode字符串编码为UTF-8：

```
In [78]: val_utf8 = val.encode('utf-8')

In [79]: val_utf8
Out[79]: b'espa\xc3\xb1ol'

In [80]: type(val_utf8)
Out[80]: bytes
```

如果你知道一个字节对象的Unicode编码，用`decode`方法可以解码：

```
In [81]: val_utf8.decode('utf-8')
Out[81]: 'español'
```

虽然UTF-8编码已经变成主流，，但因为历史的原因，你仍然可能碰到其它编码的数据：

```
In [82]: val.encode('latin1')
Out[82]: b'espa\xf1ol'

In [83]: val.encode('utf-16')
Out[83]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l

In [84]: val.encode('utf-16le')
Out[84]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

工作中碰到的文件很多都是字节对象，盲目地将所有数据编码为Unicode是不可取的。

虽然用的不多，你可以在字节文本的前面加上一个b：

```
In [85]: bytes_val = b'this is bytes'

In [86]: bytes_val
```

```
Out[86]: b'this is bytes'

In [87]: decoded = bytes_val.decode('utf8')

In [88]: decoded  # this is str (Unicode) now
Out[88]: 'this is bytes'
```

布尔值

Python中的布尔值有两个，True和False。比较和其它条件表达式可以用True和False判断。布尔值可以与and和or结合使用：

```
In [89]: True and True
Out[89]: True

In [90]: False or True
Out[90]: True
```

类型转换

str、bool、int和float也是函数，可以用来转换类型：

```
In [91]: s = '3.14159'

In [92]: fval = float(s)

In [93]: type(fval)
Out[93]: float

In [94]: int(fval)
Out[94]: 3

In [95]: bool(fval)
Out[95]: True

In [96]: bool(0)
Out[96]: False
```

None

None是Python的空值类型。如果一个函数没有明确的返回值，就会默认返回None：

```
In [97]: a = None

In [98]: a is None
Out[98]: True

In [99]: b = 5

In [100]: b is not None
Out[100]: True
```

None也常常作为函数的默认参数：

```
def add_and_maybe_multiply(a, b, c=None):
    result = a + b

    if c is not None:
        result = result * c

    return result
```

另外，None不仅是一个保留字，还是唯一的NoneType的实例：

```
In [101]: type(None)
Out[101]: NoneType
```

日期和时间

Python内建的`datetime`模块提供了`datetime`、`date`和`time`类型。`datetime`类型结合了`date`和`time`，是最常使用的：

```
In [102]: from datetime import datetime, date, time
```

```
In [103]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [104]: dt.day
Out[104]: 29

In [105]: dt.minute
Out[105]: 30
```

根据`datetime`实例，你可以用`date`和`time`提取出各自的对象：

```
In [106]: dt.date()
Out[106]: datetime.date(2011, 10, 29)

In [107]: dt.time()
Out[107]: datetime.time(20, 30, 21)
```

`strftime`方法可以将datetime格式化为字符串：

```
In [108]: dt.strftime('%m/%d/%Y %H:%M')
Out[108]: '10/29/2011 20:30'
```

`strptime`可以将字符串转换成`datetime`对象：

```
In [109]: datetime.strptime('20091031', '%Y%m%d')
Out[109]: datetime.datetime(2009, 10, 31, 0, 0)
```

表2-5列出了所有的格式化命令。

类型	说明
%Y	四位数字的年
%y	两位数字的年
%m	两位数字的月[01, 12]
%d	两位数字的天[01, 31]
%H	小时（24 小时制）[00, 23]
%I	小时（12 小时制） [01, 12]
%M	两位数字的分[00, 59]
%S	秒[00, 61]（60、61 表示闰秒）
%w	整数的周几[0 (周日), 6]
%U	第几周[00, 53]; 周日当作一周的开始，第一个周日前面的天数作为 “week 0”
%W	第几周[00, 53]; 周一当作一周的开始，第一个周一前面的天数作为 “week 0”
%z	UTC 时区偏移量为+HHMM 或-HHMM; 不知道时区则为空
%F	表示%Y-%m-%d (即 2012-4-18)
%D	表示%m/%d/%y (即 04/18/12)

表2-5 Datetime格式化指令（与ISO C89兼容）

当你聚类或对时间序列进行分组，替换datetimes的time字段有时会很有用。例如，用0替换分和秒：

```
In [110]: dt.replace(minute=0, second=0)
Out[110]: datetime.datetime(2011, 10, 29, 20, 0)
```

因为`datetime.datetime`是不可变类型，上面的方法会产生新的对象。

两个datetime对象的差会产生一个`datetime.timedelta`类型：

```
In [111]: dt2 = datetime(2011, 11, 15, 22, 30)

In [112]: delta = dt2 - dt

In [113]: delta
Out[113]: datetime.timedelta(17, 7179)

In [114]: type(delta)
Out[114]: datetime.timedelta
```

结果`timedelta(17, 7179)`指明了`timedelta`将17天、7179秒的

编码方式。

将`timedelta`添加到`datetime`，会产生一个新的偏移`datetime`：

```
In [115]: dt
Out[115]: datetime.datetime(2011, 10, 29, 20, 30, 21)

In [116]: dt + delta
Out[116]: datetime.datetime(2011, 11, 15, 22, 30)
```

控制流

Python有若干内建的关键字进行条件逻辑、循环和其它控制流操作。

if、elif和else

if是最广为人知的控制流语句。它检查一个条件，如果为True，就执行后面的语句：

```
if x < 0:
    print('It's negative')
```

`if`后面可以跟一个或多个`elif`，所有条件都是False时，还可以添加一个`else`：

```
if x < 0:
    print('It's negative')
elif x == 0:
    print('Equal to zero')
elif 0 < x < 5:
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')
```

如果某个条件为True，后面的`elif`就不会被执行。当使用and和

or时，复合条件语句是从左到右执行：

```
In [117]: a = 5; b = 7

In [118]: c = 8; d = 4

In [119]: if a < b or c > d:
   .....:     print('Made it')
Made it
```

在这个例子中，`c > d`不会被执行，因为第一个比较是True：

也可以把比较式串在一起：

```
In [120]: 4 > 3 > 2 > 1
Out[120]: True
```

for循环

for循环是在一个集合（列表或元组）中进行迭代，或者就是一个迭代器。for循环的标准语法是：

```
for value in collection:
    # do something with value
```

你可以用continue使for循环提前，跳过剩下的部分。看下面这个例子，将一个列表中的整数相加，跳过None：

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
    if value is None:
        continue
    total += value
```

可以用`break`跳出for循环。下面的代码将各元素相加，直到遇到5：

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

break只中断for循环的最内层，其余的for循环仍会运行：

```
In [121]: for i in range(4):
   .....:     for j in range(4):
   .....:         if j > i:
   .....:             break
   .....:         print((i, j))
   .....:
(0, 0)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
(2, 2)
(3, 0)
(3, 1)
(3, 2)
(3, 3)
```

如果集合或迭代器中的元素序列（元组或列表），可以用for循环将其方便地拆分成变量：

```
for a, b, c in iterator:
    # do something
```

While循环

while循环指定了条件和代码，当条件为False或用break退出循环，代码才会退出：

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

pass

pass是Python中的非操作语句。代码块不需要任何动作时可以使用（作为未执行代码的占位符）；因为Python需要使用空白字符划定代码块，所以需要pass：

```
if x < 0:
    print('negative!')
elif x == 0:
    # TODO: put something smart here
    pass
else:
    print('positive!')
```

range

range函数返回一个迭代器，它产生一个均匀分布的整数序列：

```
In [122]: range(10)
Out[122]: range(0, 10)

In [123]: list(range(10))
Out[123]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range的三个参数是（起点，终点，步进）：

```
In [124]: list(range(0, 20, 2))
Out[124]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
In [125]: list(range(5, 0, -1))
Out[125]: [5, 4, 3, 2, 1]
```

可以看到，range产生的整数不包括终点。range的常见用法是用序号迭代序列：

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

可以使用list来存储range在其他数据结构中生成的所有整数，默认的迭代器形式通常是你想要的。下面的代码对0到99999中3或5的倍数求和：

```
sum = 0
for i in range(100000):
    # % is the modulo operator
    if i % 3 == 0 or i % 5 == 0:
        sum += i
```

虽然range可以产生任意大的数，但任意时刻耗用的内存却很小。

三元表达式

Python中的三元表达式可以将if-else语句放到一行里。语法如下：

```
value = true-expr if condition else false-expr
```

`true-expr`或`false-expr`可以是任何Python代码。它和下面的代码效果相同：

```
if condition:
    value = true-expr
else:
    value = false-expr
```

下面是一个更具体的例子：

```
In [126]: x = 5

In [127]: 'Non-negative' if x >= 0 else 'Negative'
Out[127]: 'Non-negative'
```

和if-else一样，只有一个表达式会被执行。因此，三元表达式中的if和else可以包含大量的计算，但只有True的分支会被执行。

虽然使用三元表达式可以压缩代码，但会降低代码可读性。

第3章 Python的数据结构、函数和文件

本章讨论Python的内置功能，这些功能本书会用到很多。虽然扩展库，比如pandas和Numpy，使处理大数据集很方便，但它们是和Python的内置数据处理工具一同使用的。

我们会从Python最基础的数据结构开始：元组、列表、字典和集合。然后会讨论创建你自己的、可重复使用的Python函数。最后，会学习Python的文件对象，以及如何与本地硬盘交互。

3.1 数据结构和序列

Python的数据结构简单而强大。通晓它们才能成为熟练的Python程序员。

元组

元组是一个固定长度，不可改变的Python序列对象。创建元组的最简单方式，是用逗号分隔一列值：

```
In [1]: tup = 4, 5, 6

In [2]: tup
Out[2]: (4, 5, 6)
```

当用复杂的表达式定义元组，最好将值放到圆括号内，如下所示：

```
In [3]: nested_tup = (4, 5, 6), (7, 8)

In [4]: nested_tup
Out[4]: ((4, 5, 6), (7, 8))
```

用`tuple`可以将任意序列或迭代器转换成元组：

```
In [5]: tuple([4, 0, 2])
Out[5]: (4, 0, 2)
```

```
In [6]: tup = tuple('string')

In [7]: tup
Out[7]: ('s', 't', 'r', 'i', 'n', 'g')
```

可以用方括号访问元组中的元素。和C、C++、JAVA等语言一样，序列是从0开始的：

```
In [8]: tup[0]
Out[8]: 's'
```

元组中存储的对象可能是可变对象。一旦创建了元组，元组中的对象就不能修改了：

```
In [9]: tup = tuple(['foo', [1, 2], True])

In [10]: tup[2] = False
-----------------------------------------------------
TypeError                                 Traceback (
<ipython-input-10-c7308343b841> in <module>()
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assig
```

如果元组中的某个对象是可变的，比如列表，可以在原位进行修改：

```
In [11]: tup[1].append(3)

In [12]: tup
Out[12]: ('foo', [1, 2, 3], True)
```

可以用加号运算符将元组串联起来：

```
In [13]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[13]: (4, None, 'foo', 6, 0, 'bar')
```

元组乘以一个整数，像列表一样，会将几个元组的复制串联起来：

```
In [14]: ('foo', 'bar') * 4
Out[14]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', '
```

对象本身并没有被复制，只是引用了它。

拆分元组

如果你想将元组赋值给类似元组的变量，Python会试图拆分等号右边的值：

```
In [15]: tup = (4, 5, 6)

In [16]: a, b, c = tup

In [17]: b
Out[17]: 5
```

即使含有元组的元组也会被拆分：

```
In [18]: tup = 4, 5, (6, 7)

In [19]: a, b, (c, d) = tup

In [20]: d
Out[20]: 7
```

使用这个功能，你可以很容易地替换变量的名字，其它语言可能是这样：

```
tmp = a
a = b
b = tmp
```

但是在Python中，替换可以这样做：

```
In [21]: a, b = 1, 2

In [22]: a
Out[22]: 1

In [23]: b
Out[23]: 2

In [24]: b, a = a, b

In [25]: a
Out[25]: 2

In [26]: b
Out[26]: 1
```

变量拆分常用来迭代元组或列表序列：

```
In [27]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [28]: for a, b, c in seq:
   ....:     print('a={0}, b={1}, c={2}'.format(a, b,
a=1, b=2, c=3
a=4, b=5, c=6
a=7, b=8, c=9
```

另一个常见用法是从函数返回多个值。后面会详解。

Python最近新增了更多高级的元组拆分功能，允许从元组的开头“摘取”几个元素。它使用了特殊的语法`*rest`，这也用在函数签名中以抓取任意长度列表的位置参数：

```
In [29]: values = 1, 2, 3, 4, 5
```

```
In [30]: a, b, *rest = values

In [31]: a, b
Out[31]: (1, 2)

In [32]: rest
Out[32]: [3, 4, 5]
```

`rest`的部分是想要舍弃的部分，rest的名字不重要。作为惯用写法，许多Python程序员会将不需要的变量使用下划线：

```
In [33]: a, b, *_ = values
```

tuple方法

因为元组的大小和内容不能修改，它的实例方法都很轻量。其中一个很有用的就是`count`（也适用于列表），它可以统计某个值得出现频率：

```
In [34]: a = (1, 2, 2, 2, 3, 4, 2)

In [35]: a.count(2)
Out[35]: 4
```

列表

与元组对比，列表的长度可变、内容可以被修改。你可以用方括号定义，或用`list`函数：

```
In [36]: a_list = [2, 3, 7, None]

In [37]: tup = ('foo', 'bar', 'baz')

In [38]: b_list = list(tup)

In [39]: b_list
Out[39]: ['foo', 'bar', 'baz']
```

```
In [40]: b_list[1] = 'peekaboo'

In [41]: b_list
Out[41]: ['foo', 'peekaboo', 'baz']
```

列表和元组的语义接近，在许多函数中可以交叉使用。

`list`函数常用来在数据处理中实体化迭代器或生成器：

```
In [42]: gen = range(10)

In [43]: gen
Out[43]: range(0, 10)

In [44]: list(gen)
Out[44]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

添加和删除元素

可以用`append`在列表末尾添加元素：

```
In [45]: b_list.append('dwarf')

In [46]: b_list
Out[46]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

`insert`可以在特定的位置插入元素：

```
In [47]: b_list.insert(1, 'red')

In [48]: b_list
Out[48]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

插入的序号必须在0和列表长度之间。

> 警告：与`append`相比，`insert`耗费的计算量大，因为对后续元素的引用必须在内部迁移，以便为新元素提供空间。如果

> 要在序列的头部和尾部插入元素，你可能需要使用`collections.deque`，一个双尾部队列。

insert的逆运算是pop，它移除并返回指定位置的元素：

```
In [49]: b_list.pop(2)
Out[49]: 'peekaboo'

In [50]: b_list
Out[50]: ['foo', 'red', 'baz', 'dwarf']
```

可以用`remove`去除某个值，`remove`会先寻找第一个值并除去：

```
In [51]: b_list.append('foo')

In [52]: b_list
Out[52]: ['foo', 'red', 'baz', 'dwarf', 'foo']

In [53]: b_list.remove('foo')

In [54]: b_list
Out[54]: ['red', 'baz', 'dwarf', 'foo']
```

如果不考虑性能，使用`append`和`remove`，可以把Python的列表当做完美的“多重集”数据结构。

用`in`可以检查列表是否包含某个值：

```
In [55]: 'dwarf' in b_list
Out[55]: True
```

否定`in`可以再加一个not：

```
In [56]: 'dwarf' not in b_list
Out[56]: False
```

在列表中检查是否存在某个值远比字典和集合速度慢，因为

Python是线性搜索列表中的值，但在字典和集合中，在同样的时间内还可以检查其它项（基于哈希表）。

串联和组合列表

与元组类似，可以用加号将两个列表串联起来：

```
In [57]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[57]: [4, None, 'foo', 7, 8, (2, 3)]
```

如果已经定义了一个列表，用`extend`方法可以追加多个元素：

```
In [58]: x = [4, None, 'foo']

In [59]: x.extend([7, 8, (2, 3)])

In [60]: x
Out[60]: [4, None, 'foo', 7, 8, (2, 3)]
```

通过加法将列表串联的计算量较大，因为要新建一个列表，并且要复制对象。用`extend`追加元素，尤其是到一个大列表中，更为可取。因此：

```
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)
```

要比串联方法快：

```
everything = []
for chunk in list_of_lists:
    everything = everything + chunk
```

排序

你可以用`sort`函数将一个列表原地排序（不创建新的对象）：

```
In [61]: a = [7, 2, 5, 1, 3]

In [62]: a.sort()

In [63]: a
Out[63]: [1, 2, 3, 5, 7]
```

`sort`有一些选项，有时会很好用。其中之一是二级排序key，可以用这个key进行排序。例如，我们可以按长度对字符串进行排序：

```
In [64]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [65]: b.sort(key=len)

In [66]: b
Out[66]: ['He', 'saw', 'six', 'small', 'foxes']
```

稍后，我们会学习`sorted`函数，它可以产生一个排好序的序列副本。

二分搜索和维护已排序的列表

`bisect`模块支持二分查找，和向已排序的列表插入值。`bisect.bisect`可以找到插入值后仍保证排序的位置，`bisect.insort`是向这个位置插入值：

```
In [67]: import bisect

In [68]: c = [1, 2, 2, 2, 3, 4, 7]

In [69]: bisect.bisect(c, 2)
Out[69]: 4

In [70]: bisect.bisect(c, 5)
Out[70]: 6
```

```
In [71]: bisect.insort(c, 6)

In [72]: c
Out[72]: [1, 2, 2, 2, 3, 4, 6, 7]
```

> 注意：`bisect`模块不会检查列表是否已排好序，进行检查的话会耗费大量计算。因此，对未排序的列表使用`bisect`不会产生错误，但结果不一定正确。

切片

用切边可以选取大多数序列类型的一部分，切片的基本形式是在方括号中使用`start:stop`：

```
In [73]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [74]: seq[1:5]
Out[74]: [2, 3, 7, 5]
```

切片也可以被序列赋值：

```
In [75]: seq[3:4] = [6, 3]

In [76]: seq
Out[76]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

切片的起始元素是包括的，不包含结束元素。因此，结果中包含的元素个数是`stop - start`。

`start`或`stop`都可以被省略，省略之后，分别默认序列的开头和结尾：

```
In [77]: seq[:5]
Out[77]: [7, 2, 3, 6, 3]

In [78]: seq[3:]
Out[78]: [6, 3, 5, 6, 0, 1]
```

负数表明从后向前切片：

```
In [79]: seq[-4:]
Out[79]: [5, 6, 0, 1]

In [80]: seq[-6:-2]
Out[80]: [6, 3, 5, 6]
```

需要一段时间来熟悉使用切片，尤其是当你之前学的是R或MATLAB。图3-1展示了正整数和负整数的切片。在图中，指数标示在边缘以表明切片是在哪里开始哪里结束的。

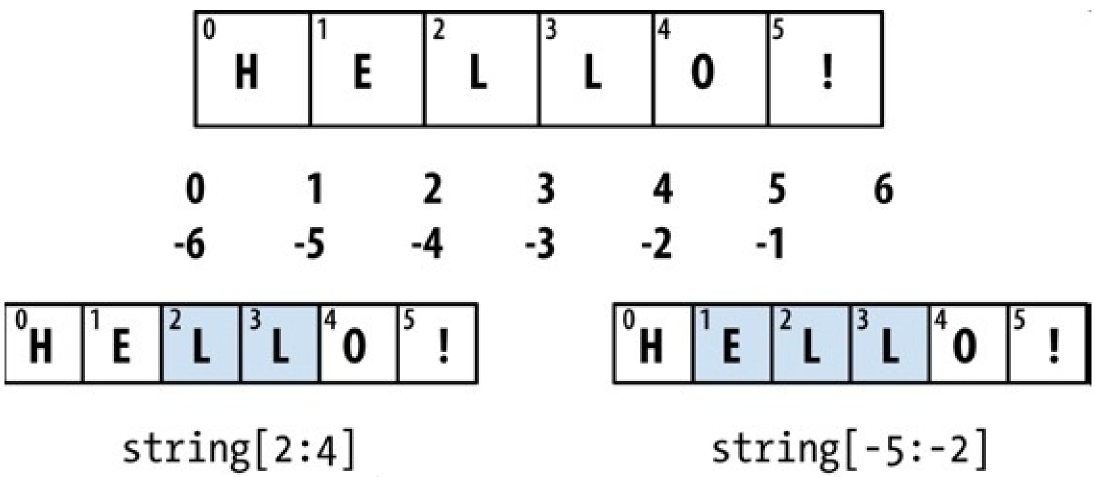


图3-1 Python切片演示

在第二个冒号后面使用`step`，可以隔一个取一个元素：

```
In [81]: seq[::2]
Out[81]: [7, 3, 3, 6, 1]
```

一个聪明的方法是使用`-1`，它可以将列表或元组颠倒过来：

```
In [82]: seq[::-1]
Out[82]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

序列函数

Python有一些有用的序列函数。

enumerate函数

迭代一个序列时，你可能想跟踪当前项的序号。手动的方法可能是下面这样：

```
i = 0
for value in collection:
   # do something with value
   i += 1
```

因为这么做很常见，Python内建了一个`enumerate`函数，可以返回`(i, value)`元组序列：

```
for i, value in enumerate(collection):
   # do something with value
```

当你索引数据时，使用`enumerate`的一个好方法是计算序列（唯一的）`dict`映射到位置的值：

```
In [83]: some_list = ['foo', 'bar', 'baz']

In [84]: mapping = {}

In [85]: for i, v in enumerate(some_list):
   ....:     mapping[v] = i

In [86]: mapping
Out[86]: {'bar': 1, 'baz': 2, 'foo': 0}
```

sorted函数

`sorted`函数可以从任意序列的元素返回一个新的排好序的列表：

```
In [87]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[87]: [0, 1, 2, 2, 3, 6, 7]

In [88]: sorted('horse race')
```

```
Out[88]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r'
```

`sorted`函数可以接受和`sort`相同的参数。

zip函数

`zip`可以将多个列表、元组或其它序列成对组合成一个元组列表：

```
In [89]: seq1 = ['foo', 'bar', 'baz']

In [90]: seq2 = ['one', 'two', 'three']

In [91]: zipped = zip(seq1, seq2)

In [92]: list(zipped)
Out[92]: [('foo', 'one'), ('bar', 'two'), ('baz', 'th
```

`zip`可以处理任意多的序列，元素的个数取决于最短的序列：

```
In [93]: seq3 = [False, True]

In [94]: list(zip(seq1, seq2, seq3))
Out[94]: [('foo', 'one', False), ('bar', 'two', True)
```

`zip`的常见用法之一是同时迭代多个序列，可能结合`enumerate`使用：

```
In [95]: for i, (a, b) in enumerate(zip(seq1, seq2)):
   ....:     print('{0}: {1}, {2}'.format(i, a, b))
   ....:
0: foo, one
1: bar, two
2: baz, three
```

给出一个“被压缩的”序列，`zip`可以被用来解压序列。也可以当作把行的列表转换为列的列表。这个方法看起来有点神奇：

```
In [96]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Cl
   ....:             ('Schilling', 'Curt')]

In [97]: first_names, last_names = zip(*pitchers)

In [98]: first_names
Out[98]: ('Nolan', 'Roger', 'Schilling')

In [99]: last_names
Out[99]: ('Ryan', 'Clemens', 'Curt')
```

reversed函数

`reversed`可以从后向前迭代一个序列：

```
In [100]: list(reversed(range(10)))
Out[100]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

要记住`reversed`是一个生成器（后面详细介绍），只有实体化（即列表或for循环）之后才能创建翻转的序列。

字典

字典可能是Python最为重要的数据结构。它更为常见的名字是哈希映射或关联数组。它是键值对的大小可变集合，键和值都是Python对象。创建字典的方法之一是使用尖括号，用冒号分隔键和值：

```
In [101]: empty_dict = {}

In [102]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4

In [103]: d1
```

```
Out[103]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

你可以像访问列表或元组中的元素一样，访问、插入或设定字典中的元素：

```
In [104]: d1[7] = 'an integer'

In [105]: d1
Out[105]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: '

In [106]: d1['b']
Out[106]: [1, 2, 3, 4]
```

你可以用检查列表和元组是否包含某个值得方法，检查字典中是否包含某个键：

```
In [107]: 'b' in d1
Out[107]: True
```

可以用`del`关键字或`pop`方法（返回值得同时删除键）删除值：

```
In [108]: d1[5] = 'some value'

In [109]: d1
Out[109]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value'}

In [110]: d1['dummy'] = 'another value'

In [111]: d1
Out[111]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
```

```
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}

In [112]: del d1[5]

In [113]: d1
Out[113]:
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 'dummy': 'another value'}

In [114]: ret = d1.pop('dummy')

In [115]: ret
Out[115]: 'another value'

In [116]: d1
Out[116]: {'a': 'some value', 'b': [1, 2, 3, 4], 7: 'a
```

`keys`和`values`是字典的键和值的迭代器方法。虽然键值对没有顺序，这两个方法可以用相同的顺序输出键和值：

```
In [117]: list(d1.keys())
Out[117]: ['a', 'b', 7]

In [118]: list(d1.values())
Out[118]: ['some value', [1, 2, 3, 4], 'an integer']
```

用`update`方法可以将一个字典与另一个融合：

```
In [119]: d1.update({'b' : 'foo', 'c' : 12})

In [120]: d1
Out[120]: {'a': 'some value', 'b': 'foo', 7: 'an inte
```

`update`方法是原地改变字典，因此任何传递给`update`的键的旧的值都会被舍弃。

用序列创建字典

常常，你可能想将两个序列配对组合成字典。下面是一种写法：

```
mapping = {}
for key, value in zip(key_list, value_list):
    mapping[key] = value
```

因为字典本质上是2元元组的集合，dict可以接受2元元组的列表：

```
In [121]: mapping = dict(zip(range(5), reversed(range

In [122]: mapping
Out[122]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

后面会谈到`dict comprehensions`，另一种构建字典的优雅方式。

默认值

下面的逻辑很常见：

```
if key in some_dict:
    value = some_dict[key]
else:
    value = default_value
```

因此，dict的方法get和pop可以取默认值进行返回，上面的if-else语句可以简写成下面：

```
value = some_dict.get(key, default_value)
```

get默认会返回None，如果不存在键，pop会抛出一个例外。关于设定值，常见的情况是在字典的值是属于其它集合，如列表。例如，你可以通过首字母，将一个列表中的单词分类：

```
In [123]: words = ['apple', 'bat', 'bar', 'atom', 'boo

In [124]: by_letter = {}

In [125]: for word in words:
   .....:     letter = word[0]
   .....:     if letter not in by_letter:
   .....:         by_letter[letter] = [word]
   .....:     else:
   .....:         by_letter[letter].append(word)
   .....:

In [126]: by_letter
Out[126]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar'
```

`setdefault`方法就正是干这个的。前面的for循环可以改写为：

```
for word in words:
    letter = word[0]
    by_letter.setdefault(letter, []).append(word)
```

`collections`模块有一个很有用的类，`defaultdict`，它可以进一步简化上面。传递类型或函数以生成每个位置的默认值：

```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
    by_letter[word[0]].append(word)
```

有效的键类型

字典的值可以是任意Python对象，而键通常是不可变的标量类型

（整数、浮点型、字符串）或元组（元组中的对象必须是不可变的）。这被称为“可哈希性”。可以用`hash`函数检测一个对象是否是可哈希的（可被用作字典的键）：

```
In [127]: hash('string')
Out[127]: 5023931463650008331

In [128]: hash((1, 2, (2, 3)))
Out[128]: 1097636502276347782

In [129]: hash((1, 2, [2, 3])) # fails because lists a
---------------------------------------------------------
TypeError                                 Traceback (m
<ipython-input-129-800cd14ba8be> in <module>()
----> 1 hash((1, 2, [2, 3])) # fails because lists are
TypeError: unhashable type: 'list'
```

要用列表当做键，一种方法是将列表转化为元组，只要内部元素可以被哈希，它也就可以被哈希：

```
In [130]: d = {}

In [131]: d[tuple([1, 2, 3])] = 5

In [132]: d
Out[132]: {(1, 2, 3): 5}
```

集合

集合是无序的不可重复的元素的集合。你可以把它当做字典，但是只有键没有值。可以用两种方式创建集合：通过set函数或使用尖括号set语句：

```
In [133]: set([2, 2, 2, 1, 3, 3])
Out[133]: {1, 2, 3}
```

```
In [134]: {2, 2, 2, 1, 3, 3}
Out[134]: {1, 2, 3}
```

集合支持合并、交集、差分和对称差等数学集合运算。考虑两个示例集合：

```
In [135]: a = {1, 2, 3, 4, 5}

In [136]: b = {3, 4, 5, 6, 7, 8}
```

合并是取两个集合中不重复的元素。可以用`union`方法，或者`|`运算符：

```
In [137]: a.union(b)
Out[137]: {1, 2, 3, 4, 5, 6, 7, 8}

In [138]: a | b
Out[138]: {1, 2, 3, 4, 5, 6, 7, 8}
```

交集的元素包含在两个集合中。可以用`intersection`或`&`运算符：

```
In [139]: a.intersection(b)
Out[139]: {3, 4, 5}

In [140]: a & b
Out[140]: {3, 4, 5}
```

表3-1列出了常用的集合方法。

函数	替代语法	说明
a.add(x)	N/A	将元素 x 添加到集合 a
a.clear()	N/A	将集合清空
a.remove(x)	N/A	将元素 x 从集合 a 除去
a.pop()	N/A	从集合 a 去除任意元素，如果集合为空，则抛出 KeyError 错误
a.union(b)	a | b	集合 a 和 b 中的所有不重复元素
a.update(b)	a |= b	设定集合 a 中的元素为 a 与 b 的合并
a.intersection(b)	a & b	a 和 b 中交叉的元素
a.intersection_update(b)	a &= b	设定集合 a 中的元素为 a 与 b 的交叉
a.difference(b)	a - b	存在于 a 但不存在于 b 的元素
a.difference_update(b)	a -= b	设定集合 a 中的元素为 a 与 b 的差
a.symmetric_difference(b)	a ^ b	只在 a 或只在 b 的元素
a.symmetric_difference_update(b)	a ^= b	集合 a 中的元素为只在 a 或只在 b 的元素
a.issubset(b)	N/A	如果 a 中的元素全部属于 b，则为 True
a.issuperset(b)	N/A	如果 b 中的元素全部属于 a，则为 True
a.isdisjoint(b)	N/A	如果 a 和 b 无共同元素，则为 True

表3-1 Python的集合操作

所有逻辑集合操作都有另外原地实现方法，它可以直接用结果替代集合的内容。对于大的集合，这么做效率更高：

```
In [141]: c = a.copy()

In [142]: c |= b

In [143]: c
Out[143]: {1, 2, 3, 4, 5, 6, 7, 8}

In [144]: d = a.copy()

In [145]: d &= b

In [146]: d
Out[146]: {3, 4, 5}
```

与字典类似，集合元素通常都是不可变的。要获得类似列表的元素，必须转换成元组：

```
In [147]: my_data = [1, 2, 3, 4]

In [148]: my_set = {tuple(my_data)}

In [149]: my_set
Out[149]: {(1, 2, 3, 4)}
```

你还可以检测一个集合是否是另一个集合的子集或父集：

```
In [150]: a_set = {1, 2, 3, 4, 5}

In [151]: {1, 2, 3}.issubset(a_set)
Out[151]: True

In [152]: a_set.issuperset({1, 2, 3})
Out[152]: True
```

集合的内容相同时，集合才对等：

```
In [153]: {1, 2, 3} == {3, 2, 1}
Out[153]: True
```

列表、集合和字典推导式

列表推导式是Python最受喜爱的特性之一。它允许用户方便的从一个集合过滤元素，形成列表，在传递参数的过程中还可以修改元素。形式如下：

```
[expr for val in collection if condition]
```

它等同于下面的for循环;

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

filter条件可以被忽略，只留下表达式就行。例如，给定一个字符串列表，我们可以过滤出长度在2及以下的字符串，并将其转换成大写：

```
In [154]: strings = ['a', 'as', 'bat', 'car', 'dove',

In [155]: [x.upper() for x in strings if len(x) > 2]
Out[155]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

用相似的方法，还可以推导集合和字典。字典的推导式如下所示：

```
dict_comp = {key-expr : value-expr for value in colle
```

集合的推导式与列表很像，只不过用的是尖括号：

```
set_comp = {expr for value in collection if condition
```

与列表推导式类似，集合与字典的推导也很方便，而且使代码的读写都很容易。来看前面的字符串列表。假如我们只想要字符串的长度，用集合推导式的方法非常方便：

```
In [156]: unique_lengths = {len(x) for x in strings}

In [157]: unique_lengths
Out[157]: {1, 2, 3, 4, 6}
```

`map`函数可以进一步简化：

```
In [158]: set(map(len, strings))
Out[158]: {1, 2, 3, 4, 6}
```

作为一个字典推导式的例子，我们可以创建一个字符串的查找映射表以确定它在列表中的位置：

```
In [159]: loc_mapping = {val : index for index, val i

In [160]: loc_mapping
Out[160]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove
```

嵌套列表推导式

假设我们有一个包含列表的列表，包含了一些英文名和西班牙名：

```
In [161]: all_data = [['John', 'Emily', 'Michael', 'Ma
   .....:              ['Maria', 'Juan', 'Javier', 'Na
```

你可能是从一些文件得到的这些名字，然后想按照语言进行分类。现在假设我们想用一个列表包含所有的名字，这些名字中包含两个或更多的e。可以用for循环来做：

```
names_of_interest = []
for names in all_data:
    enough_es = [name for name in names if name.count
    names_of_interest.extend(enough_es)
```

可以用嵌套列表推导式的方法，将这些写在一起，如下所示：

```
In [162]: result = [name for names in all_data for na
   .....:            if name.count('e') >= 2]

In [163]: result
Out[163]: ['Steven']
```

嵌套列表推导式看起来有些复杂。列表推导式的for部分是根据嵌套的顺序，过滤条件还是放在最后。下面是另一个例子，我们将一个整数元组的列表扁平化成了一个整数列表：

```
In [164]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8,

In [165]: flattened = [x for tup in some_tuples for x

In [166]: flattened
Out[166]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

记住，for表达式的顺序是与嵌套for循环的顺序一样（而不是列表推导式的顺序）：

```
flattened = []

for tup in some_tuples:
    for x in tup:
        flattened.append(x)
```

你可以有任意多级别的嵌套，但是如果你有两三个以上的嵌套，你就应该考虑下代码可读性的问题了。分辨列表推导式的列表推导式中的语法也是很重要的：

```
In [167]: [[x for x in tup] for tup in some_tuples]
Out[167]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

这段代码产生了一个列表的列表，而不是扁平化的只包含元素的列表。

3.2 函数

函数是Python中最主要也是最重要的代码组织和复用手段。作为最重要的原则，如果你要重复使用相同或非常类似的代码，就需要写一个函数。通过给函数起一个名字，还可以提高代码的可读性。

函数使用`def`关键字声明，用`return`关键字返回值：

```
def my_function(x, y, z=1.5):
```

```
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

同时拥有多条return语句也是可以的。如果到达函数末尾时没有遇到任何一条return语句，则返回None。

函数可以有一些位置参数（positional）和一些关键字参数（keyword）。关键字参数通常用于指定默认值或可选参数。在上面的函数中，x和y是位置参数，而z则是关键字参数。也就是说，该函数可以下面这两种方式进行调用：

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
my_function(10, 20)
```

函数参数的主要限制在于：关键字参数必须位于位置参数（如果有的话）之后。你可以任何顺序指定关键字参数。也就是说，你不用死记硬背函数参数的顺序，只要记得它们的名字就可以了。

> 笔记：也可以用关键字传递位置参数。前面的例子，也可以写为：
>
> ```
> my_function(x=5, y=6, z=7)
> my_function(y=6, x=5, z=7)
> ```
>
> 这种写法可以提高可读性。

命名空间、作用域，和局部函数

函数可以访问两种不同作用域中的变量：全局（global）和局部（local）。Python有一种更科学的用于描述变量作用域的名称，即命名空间（namespace）。任何在函数中赋值的变量默认都是被分配到局部命名空间（local namespace）中的。局部命名空间是在函数被调用时创建的，函数参数会立即填入该命名空间。

在函数执行完毕之后，局部命名空间就会被销毁（会有一些例外的情况，具体请参见后面介绍闭包的那一节）。看看下面这个函数：

```
def func():
    a = []
    for i in range(5):
        a.append(i)
```

调用func()之后，首先会创建出空列表a，然后添加5个元素，最后a会在该函数退出的时候被销毁。假如我们像下面这样定义a：

```
a = []
def func():
    for i in range(5):
        a.append(i)
```

虽然可以在函数中对全局变量进行赋值操作，但是那些变量必须用global关键字声明成全局的才行：

```
In [168]: a = None

In [169]: def bind_a_variable():
   .....:     global a
   .....:     a = []
   .....: bind_a_variable()
   .....:

In [170]: print(a)
[]
```

> 注意：我常常建议人们不要频繁使用global关键字。因为全局变量一般是用于存放系统的某些状态的。如果你发现自己用了很多，那可能就说明得要来点儿面向对象编程了（即使用类）。

返回多个值

在我第一次用Python编程时（之前已经习惯了Java和C++），最喜欢的一个功能是：函数可以返回多个值。下面是一个简单的例子：

```
def f():
    a = 5
    b = 6
    c = 7
    return a, b, c

a, b, c = f()
```

在数据分析和其他科学计算应用中，你会发现自己常常这么干。该函数其实只返回了一个对象，也就是一个元组，最后该元组会被拆包到各个结果变量中。在上面的例子中，我们还可以这样写：

```
return_value = f()
```

这里的return_value将会是一个含有3个返回值的三元元组。此外，还有一种非常具有吸引力的多值返回方式——返回字典：

```
def f():
    a = 5
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

取决于工作内容，第二种方法可能很有用。

函数也是对象

由于Python函数都是对象，因此，在其他语言中较难表达的一些设计思想在Python中就要简单很多了。假设我们有下面这样一个字符串数组，希望对其进行一些数据清理工作并执行一堆转换：

```
In [171]: states = ['   Alabama ', 'Georgia!', 'Georg
   .....:          'south   carolina##', 'West virgi
```

不管是谁，只要处理过由用户提交的调查数据，就能明白这种乱七八糟的数据是怎么一回事。为了得到一组能用于分析工作的格式统一的字符串，需要做很多事情：去除空白符、删除各种标点符号、正确的大写格式等。做法之一是使用内建的字符串方法和正则表达式`re`模块：

```
import re

def clean_strings(strings):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub('[!#?]', '', value)
        value = value.title()
        result.append(value)
    return result
```

结果如下所示：

```
In [173]: clean_strings(states)
Out[173]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South   Carolina',
 'West Virginia']
```

其实还有另外一种不错的办法：将需要在一组给定字符串上执行的所有运算做成一个列表：

```
def remove_punctuation(value):
```

```
    return re.sub('[!#?]', '', value)

clean_ops = [str.strip, remove_punctuation, str.title

def clean_strings(strings, ops):
    result = []
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
```

然后我们就有了：

```
In [175]: clean_strings(states, clean_ops)
Out[175]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South   Carolina',
 'West Virginia']
```

这种多函数模式使你能在很高的层次上轻松修改字符串的转换方式。此时的clean_strings也更具可复用性！

还可以将函数用作其他函数的参数，比如内置的map函数，它用于在一组数据上应用一个函数：

```
In [176]: for x in map(remove_punctuation, states):
   .....:     print(x)
Alabama
Georgia
Georgia
georgia
```

```
FlOrIda
south   carolina
West virginia
```

匿名（lambda）函数

Python支持一种被称为匿名的、或lambda函数。它仅由单条语句组成，该语句的结果就是返回值。它是通过lambda关键字定义的，这个关键字没有别的含义，仅仅是说“我们正在声明的是一个匿名函数”。

```
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

本书其余部分一般将其称为lambda函数。它们在数据分析工作中非常方便，因为你会发现很多数据转换函数都以函数作为参数的。直接传入lambda函数比编写完整函数声明要少输入很多字（也更清晰），甚至比将lambda函数赋值给一个变量还要少输入很多字。看看下面这个简单得有些傻的例子：

```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

虽然你可以直接编写[x *2for x in ints]，但是这里我们可以非常轻松地传入一个自定义运算给apply_to_list函数。

再来看另外一个例子。假设有一组字符串，你想要根据各字符串不同字母的数量对其进行排序：

```
In [177]: strings = ['foo', 'card', 'bar', 'aaaa', 'a
```

这里，我们可以传入一个lambda函数到列表的sort方法：

```
In [178]: strings.sort(key=lambda x: len(set(list(x))

In [179]: strings
Out[179]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

> 笔记：lambda函数之所以会被称为匿名函数，与def声明的函数不同，原因之一就是这种函数对象本身是没有提供名称**name**属性。

柯里化：部分参数应用

柯里化（currying）是一个有趣的计算机科学术语，它指的是通过“部分参数应用”（partial argument application）从现有函数派生出新函数的技术。例如，假设我们有一个执行两数相加的简单函数：

```
def add_numbers(x, y):
    return x + y
```

通过这个函数，我们可以派生出一个新的只有一个参数的函数——add_five，它用于对其参数加5：

```
add_five = lambda y: add_numbers(5, y)
```

add_numbers的第二个参数称为“柯里化的”（curried）。这里没什么特别花哨的东西，因为我们其实就只是定义了一个可以调用现有函数的新函数而已。内置的functools模块可以用partial函数将此过程简化：

```
from functools import partial
add_five = partial(add_numbers, 5)
```

生成器

能以一种一致的方式对序列进行迭代（比如列表中的对象或文件中的行）是Python的一个重要特点。这是通过一种叫做迭代器协议（iterator protocol，它是一种使对象可迭代的通用方式）的方式实现的，一个原生的使对象可迭代的方法。比如说，对字典进行迭代可以得到其所有的键：

```
In [180]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [181]: for key in some_dict:
   .....:     print(key)
a
b
c
```

当你编写for key in some_dict时，Python解释器首先会尝试从some_dict创建一个迭代器：

```
In [182]: dict_iterator = iter(some_dict)

In [183]: dict_iterator
Out[183]: <dict_keyiterator at 0x7fbbd5a9f908>
```

迭代器是一种特殊对象，它可以在诸如for循环之类的上下文中向Python解释器输送对象。大部分能接受列表之类的对象的方法也都可以接受任何可迭代对象。比如min、max、sum等内置方法以及list、tuple等类型构造器：

```
In [184]: list(dict_iterator)
Out[184]: ['a', 'b', 'c']
```

生成器（generator）是构造新的可迭代对象的一种简单方式。一般的函数执行之后只会返回单个值，而生成器则是以延迟的方式返回一个值序列，即每返回一个值之后暂停，直到下一个值被请求时再继续。要创建一个生成器，只需将函数中的return替换为yeild即可：

```
def squares(n=10):
    print('Generating squares from 1 to {0}'.format(n
    for i in range(1, n + 1):
        yield i ** 2
```

调用该生成器时，没有任何代码会被立即执行：

```
In [186]: gen = squares()

In [187]: gen
Out[187]: <generator object squares at 0x7fbbd5ab4570
```

直到你从该生成器中请求元素时，它才会开始执行其代码：

```
In [188]: for x in gen:
   .....:     print(x, end=' ')
Generating squares from 1 to 100
1 4 9 16 25 36 49 64 81 100
```

生成器表达式

另一种更简洁的构造生成器的方法是使用生成器表达式（generator expression）。这是一种类似于列表、字典、集合推导式的生成器。其创建方式为，把列表推导式两端的方括号改成圆括号：

```
In [189]: gen = (x ** 2 for x in range(100))

In [190]: gen
Out[190]: <generator object <genexpr> at 0x7fbbd5ab29
```

它跟下面这个冗长得多的生成器是完全等价的：

```
def _make_gen():
    for x in range(100):
```

```
        yield x ** 2
gen = _make_gen()
```

生成器表达式也可以取代列表推导式，作为函数参数：

```
In [191]: sum(x ** 2 for x in range(100))
Out[191]: 328350

In [192]: dict((i, i **2) for i in range(5))
Out[192]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

itertools模块

标准库itertools模块中有一组用于许多常见数据算法的生成器。例如，groupby可以接受任何序列和一个函数。它根据函数的返回值对序列中的连续元素进行分组。下面是一个例子：

```
In [193]: import itertools

In [194]: first_letter = lambda x: x[0]

In [195]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Al

In [196]: for letter, names in itertools.groupby(name
   .....:     print(letter, list(names)) # names is a
A ['Alan', 'Adam']
W ['Wes', 'Will']
A ['Albert']
S ['Steven']
```

表3-2中列出了一些我经常用到的itertools函数。建议参阅Python官方文档，进一步学习。

函数	说明
combinations(iterable, k)	生成一个由 iterable 中所有可能的 k 元元组组成的序列，不考虑顺序（参阅另一个函数 combinations_with_replacement）
permutations(iterable, k)	生成一个由 iterable 中所有可能的 k 元元组组成的序列，考虑顺序
groupby(iterable[, keyfunc])	为每个唯一键生成一个（key, sub-iterator）
product(*iterables, repeat=1)	生成输入的 iterables 的笛卡尔积，结果为元组，类似于嵌套 for 循环

表3-2 一些有用的itertools函数

错误和异常处理

优雅地处理Python的错误和异常是构建健壮程序的重要部分。在数据分析中，许多函数函数只用于部分输入。例如，Python的float函数可以将字符串转换成浮点数，但输入有误时，有`ValueError`错误：

```
In [197]: float('1.2345')
Out[197]: 1.2345

In [198]: float('something')
---------------------------------------------------------------
ValueError                                Traceback (
<ipython-input-198-439904410854> in <module>()
----> 1 float('something')
ValueError: could not convert string to float: 'somet
```

假如想优雅地处理float的错误，让它返回输入值。我们可以写一个函数，在try/except中调用float：

```
def attempt_float(x):
    try:
        return float(x)
    except:
        return x
```

当float(x)抛出异常时，才会执行except的部分：

```
In [200]: attempt_float('1.2345')
Out[200]: 1.2345

In [201]: attempt_float('something')
Out[201]: 'something'
```

你可能注意到float抛出的异常不仅是ValueError：

```
In [202]: float((1, 2))
---------------------------------------------------------
TypeError                                 Traceback (
<ipython-input-202-842079ebb635> in <module>()
----> 1 float((1, 2))
TypeError: float() argument must be a string or a num
```

你可能只想处理ValueError，TypeError错误（输入不是字符串或数值）可能是合理的bug。可以写一个异常类型：

```
def attempt_float(x):
    try:
        return float(x)
    except ValueError:
        return x
```

然后有：

```
In [204]: attempt_float((1, 2))
---------------------------------------------------------
TypeError                                 Traceback (
<ipython-input-204-9bdfd730cead> in <module>()
----> 1 attempt_float((1, 2))
<ipython-input-203-3e06b8379b6b> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
```

```
----> 3         return float(x)
      4     except ValueError:
      5         return x
TypeError: float() argument must be a string or a numl
```

可以用元组包含多个异常：

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError):
        return x
```

某些情况下，你可能不想抑制异常，你想无论try部分的代码是否成功，都执行一段代码。可以使用finally：

```
f = open(path, 'w')

try:
    write_to_file(f)
finally:
    f.close()
```

这里，文件处理f总会被关闭。相似的，你可以用else让只在try部分成功的情况下，才执行代码：

```
f = open(path, 'w')

try:
    write_to_file(f)
except:
    print('Failed')
else:
    print('Succeeded')
finally:
    f.close()
```

IPython的异常

如果是在%run一个脚本或一条语句时抛出异常，IPython默认会打印完整的调用栈（traceback），在栈的每个点都会有几行上下文：

```
In [10]: %run examples/ipython_bug.py
---------------------------------------------------------------
AssertionError                            Traceback (m
/home/wesm/code/pydata-book/examples/ipython_bug.py in
     13     throws_an_exception()
     14
---> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in
     11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py in
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:
```

自身就带有文本是相对于Python标准解释器的极大优点。你可以用魔术命令`%xmode`，从Plain（与Python标准解释器相同）到Verbose（带有函数的参数值）控制文本显示的数量。后面可以看到，发生错误之后，（用%debug或%pdb magics）可以进入stack进行事后调试。

3.3 文件和操作系统

本书的代码示例大多使用诸如pandas.read_csv之类的高级工具将磁盘上的数据文件读入Python数据结构。但我们还是需要了解一些有关Python文件处理方面的基础知识。好在它本来就很简单，这也是Python在文本和文件处理方面的如此流行的原因之一。

为了打开一个文件以便读写，可以使用内置的open函数以及一个相对或绝对的文件路径：

```
In [207]: path = 'examples/segismundo.txt'

In [208]: f = open(path)
```

默认情况下，文件是以只读模式（'r'）打开的。然后，我们就可以像处理列表那样来处理这个文件句柄f了，比如对行进行迭代：

```
for line in f:
    pass
```

从文件中取出的行都带有完整的行结束符（EOL），因此你常常会看到下面这样的代码（得到一组没有EOL的行）：

```
In [209]: lines = [x.rstrip() for x in open(path)]

In [210]: lines
Out[210]:
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sueña el que a medrar empieza,',
 'sueña el que afana y pretende,',
```

```
  'sueña el que agravia y ofende,',
  '',
  'y en el mundo, en conclusión,',
  'todos sueñan lo que son,',
  'aunque ninguno lo entiende.',
  '']
```

如果使用open创建文件对象，一定要用close关闭它。关闭文件可以返回操作系统资源：

```
In [211]: f.close()
```

用with语句可以可以更容易地清理打开的文件：

```
In [212]: with open(path) as f:
   .....:     lines = [x.rstrip() for x in f]
```

这样可以在退出代码块时，自动关闭文件。

如果输入f =open(path,'w')，就会有一个新文件被创建在examples/segismundo.txt，并覆盖掉该位置原来的任何数据。另外有一个x文件模式，它可以创建可写的文件，但是如果文件路径存在，就无法创建。表3-3列出了所有的读/写模式。

模式	说明
r	只读模式
w	只写模式。创建新文件（删除同名的任何文件[译注12]）
a	附加到现有文件（如果文件不存在则创建一个）
r+	读写模式
b	附加说明某模式用于二进制文件，即'rb'或'wb'
U	通用换行模式。单独使用'U'或附加到其他读模式（如'rU'）

表3-3 Python的文件模式

对于可读文件，一些常用的方法是read、seek和tell。read会从文件返回字符。字符的内容是由文件的编码决定的（如UTF-8），如果是二进制模式打开的就是原始字节：

```
In [213]: f = open(path)

In [214]: f.read(10)
Out[214]: 'Sueña el r'

In [215]: f2 = open(path, 'rb')  # Binary mode

In [216]: f2.read(10)
Out[216]: b'Sue\xc3\xb1a el '
```

read模式会将文件句柄的位置提前，提前的数量是读取的字节数。tell可以给出当前的位置：

```
In [217]: f.tell()
Out[217]: 11

In [218]: f2.tell()
Out[218]: 10
```

尽管我们从文件读取了10个字符，位置却是11，这是因为用默认的编码用了这么多字节才解码了这10个字符。你可以用sys模块检查默认的编码：

```
In [219]: import sys

In [220]: sys.getdefaultencoding()
Out[220]: 'utf-8'
```

seek将文件位置更改为文件中的指定字节：

```
In [221]: f.seek(3)
Out[221]: 3

In [222]: f.read(1)
Out[222]: 'ñ'
```

最后，关闭文件：

```
In [223]: f.close()

In [224]: f2.close()
```

向文件写入，可以使用文件的write或writelines方法。例如，我们可以创建一个无空行版的prof_mod.py：

```
In [225]: with open('tmp.txt', 'w') as handle:
   .....:     handle.writelines(x for x in open(path)

In [226]: with open('tmp.txt') as f:
   .....:     lines = f.readlines()

In [227]: lines
Out[227]:
['Sueña el rico en su riqueza,\n',
 'que más cuidados le ofrece;\n',
 'sueña el pobre que padece\n',
 'su miseria y su pobreza;\n',
 'sueña el que a medrar empieza,\n',
 'sueña el que afana y pretende,\n',
 'sueña el que agravia y ofende,\n',
 'y en el mundo, en conclusión,\n',
 'todos sueñan lo que son,\n',
 'aunque ninguno lo entiende.\n']
```

表3-4列出了一些最常用的文件方法。

方法	说明
read([size])	以字符串形式返回文件数据，可选的size参数用于说明读取的字节数
readlines([size])	将文件返回为行列表，可选参数size
write(str)	将字符串写入文件
close()	关闭句柄
flush()	清空内部I/O缓存区，并将数据强行写回磁盘
seek(pos)	移动到指定的文件位置（整数）
tell()	以整数形式返回当前文件位置
closed	如果文件已关闭，则为True

表3-4 Python重要的文件方法或属性

文件的字节和Unicode

Python文件的默认操作是“文本模式”，也就是说，你需要处理Python的字符串（即Unicode）。它与“二进制模式”相对，文件模式加一个b。我们来看上一节的文件（UTF-8编码、包含非ASCII字符）：

```
In [230]: with open(path) as f:
   .....:     chars = f.read(10)

In [231]: chars
Out[231]: 'Sueña el r'
```

UTF-8是长度可变的Unicode编码，所以当我从文件请求一定数量的字符时，Python会从文件读取足够多（可能少至10或多至40字节）的字节进行解码。如果以“rb”模式打开文件，则读取确切的请求字节数：

```
In [232]: with open(path, 'rb') as f:
   .....:     data = f.read(10)

In [233]: data
Out[233]: b'Sue\xc3\xb1a el '
```

取决于文本的编码，你可以将字节解码为str对象，但只有当每个

编码的Unicode字符都完全成形时才能这么做：

```
In [234]: data.decode('utf8')
Out[234]: 'Sueña el '

In [235]: data[:4].decode('utf8')
-----------------------------------------------------
UnicodeDecodeError                        Traceback (
<ipython-input-235-300e0af10bb7> in <module>()
----> 1 data[:4].decode('utf8')
UnicodeDecodeError: 'utf-8' codec can't decode byte 0
d end of data
```

文本模式结合了open的编码选项，提供了一种更方便的方法将Unicode转换为另一种编码：

```
In [236]: sink_path = 'sink.txt'

In [237]: with open(path) as source:
   .....:     with open(sink_path, 'xt', encoding='is
   .....:         sink.write(source.read())

In [238]: with open(sink_path, encoding='iso-8859-1')
   .....:     print(f.read(10))
Sueña el r
```

注意，不要在二进制模式中使用seek。如果文件位置位于定义Unicode字符的字节的中间位置，读取后面会产生错误：

```
In [240]: f = open(path)

In [241]: f.read(5)
Out[241]: 'Sueña'

In [242]: f.seek(4)
```

```
Out[242]: 4

In [243]: f.read(1)
------------------------------------------------------------------
UnicodeDecodeError                               Traceback (m
<ipython-input-243-7841103e33f5> in <module>()
----> 1 f.read(1)
/miniconda/envs/book-env/lib/python3.6/codecs.py in de
    319         # decode input (taking the buffer into
    320         data = self.buffer + input
--> 321         (result, consumed) = self._buffer_deco
)
    322         # keep undecoded input until the next
    323         self.buffer = data[consumed:]
UnicodeDecodeError: 'utf-8' codec can't decode byte 0x
tart byte

In [244]: f.close()
```

如果你经常要对非ASCII字符文本进行数据分析，通晓Python的Unicode功能是非常重要的。更多内容，参阅Python官方文档。

3.4 结论

我们已经学过了Python的基础、环境和语法，接下来学习NumPy和Python的面向数组计算。

第4章 NumPy基础：数组和矢量计算

NumPy（Numerical Python的简称）是Python数值计算最重要的基础包。大多数提供科学计算的包都是用NumPy的数组作为构建基础。

NumPy的部分功能如下：

- ndarray，一个具有矢量算术运算和复杂广播能力的快速且节省空间的多维数组。
- 用于对整组数据进行快速运算的标准数学函数（无需编写循环）。
- 用于读写磁盘数据的工具以及用于操作内存映射文件的工具。
- 线性代数、随机数生成以及傅里叶变换功能。
- 用于集成由C、C++、Fortran等语言编写的代码的A C API。

由于NumPy提供了一个简单易用的C API，因此很容易将数据传递给由低级语言编写的外部库，外部库也能以NumPy数组的形式将数据返回给Python。这个功能使Python成为一种包装C/C++/Fortran历史代码库的选择，并使被包装库拥有一个动态的、易用的接口。

NumPy本身并没有提供多么高级的数据分析功能，理解NumPy数组以及面向数组的计算将有助于你更加高效地使用诸如pandas之类的工具。因为NumPy是一个很大的题目，我会在附录A中介绍更多NumPy高级功能，比如广播。

对于大部分数据分析应用而言，我最关注的功能主要集中在：

- 用于数据整理和清理、子集构造和过滤、转换等快速的矢量化数组运算。
- 常用的数组算法，如排序、唯一化、集合运算等。
- 高效的描述统计和数据聚合/摘要运算。
- 用于异构数据集的合并/连接运算的数据对齐和关系型数据运算。

- 将条件逻辑表述为数组表达式（而不是带有if-elif-else分支的循环）。
- 数据的分组运算（聚合、转换、函数应用等）。。

虽然NumPy提供了通用的数值数据处理的计算基础，但大多数读者可能还是想将pandas作为统计和分析工作的基础，尤其是处理表格数据时。pandas还提供了一些NumPy所没有的更加领域特定的功能，如时间序列处理等。

> 笔记：Python的面向数组计算可以追溯到1995年，Jim Hugunin创建了Numeric库。接下来的10年，许多科学编程社区纷纷开始使用Python的数组编程，但是进入21世纪，库的生态系统变得碎片化了。2005年，Travis Oliphant从Numeric和Numarray项目整了出了NumPy项目，进而所有社区都集合到了这个框架下。

NumPy之于数值计算特别重要的原因之一，是因为它可以高效处理大数组的数据。这是因为：

- NumPy是在一个连续的内存块中存储数据，独立于其他Python内置对象。NumPy的C语言编写的算法库可以操作内存，而不必进行类型检查或其它前期工作。比起Python的内置序列，NumPy数组使用的内存更少。
- NumPy可以在整个数组上执行复杂的计算，而不需要Python的for循环。

要搞明白具体的性能差距，考察一个包含一百万整数的数组，和一个等价的Python列表：

```
In [7]: import numpy as np

In [8]: my_arr = np.arange(1000000)

In [9]: my_list = list(range(1000000))
```

各个序列分别乘以2：

```
In [10]: %time for _ in range(10): my_arr2 = my_arr *
CPU times: user 20 ms, sys: 50 ms, total: 70 ms
Wall time: 72.4 ms

In [11]: %time for _ in range(10): my_list2 = [x * 2
CPU times: user 760 ms, sys: 290 ms, total: 1.05 s
Wall time: 1.05 s
```

基于NumPy的算法要比纯Python快10到100倍（甚至更快），并且使用的内存更少。

4.1 NumPy的ndarray：一种多维数组对象

NumPy最重要的一个特点就是其N维数组对象（即ndarray），该对象是一个快速而灵活的大数据集容器。你可以利用这种数组对整块数据执行一些数学运算，其语法跟标量元素之间的运算一样。

要明白Python是如何利用与标量值类似的语法进行批次计算，我先引入NumPy，然后生成一个包含随机数据的小数组：

```
In [12]: import numpy as np

# Generate some random data
In [13]: data = np.random.randn(2, 3)

In [14]: data
Out[14]:
array([[-0.2047,  0.4789, -0.5194],
       [-0.5557,  1.9658,  1.3934]])
```

然后进行数学运算：

```
In [15]: data * 10
Out[15]:
array([[ -2.0471,   4.7894,  -5.1944],
```

```
       [ -5.5573,  19.6578,  13.9341]])

In [16]: data + data
Out[16]:
array([[-0.4094,  0.9579, -1.0389],
       [-1.1115,  3.9316,  2.7868]])
```

第一个例子中，所有的元素都乘以10。第二个例子中，每个元素都与自身相加。

> 笔记：在本章及全书中，我会使用标准的NumPy惯用法`import numpy as np`。你当然也可以在代码中使用`from numpy import *`，但不建议这么做。`numpy`的命名空间很大，包含许多函数，其中一些的名字与Python的内置函数重名（比如min和max）。

ndarray是一个通用的同构数据多维容器，也就是说，其中的所有元素必须是相同类型的。每个数组都有一个shape（一个表示各维度大小的元组）和一个dtype（一个用于说明数组数据类型的对象）：

```
In [17]: data.shape
Out[17]: (2, 3)

In [18]: data.dtype
Out[18]: dtype('float64')
```

本章将会介绍NumPy数组的基本用法，这对于本书后面各章的理解基本够用。虽然大多数数据分析工作不需要深入理解NumPy，但是精通面向数组的编程和思维方式是成为Python科学计算牛人的一大关键步骤。

> 笔记：当你在本书中看到“数组”、“NumPy数组”、"ndarray"时，基本上都指的是同一样东西，即ndarray对象。

创建ndarray

创建数组最简单的办法就是使用array函数。它接受一切序列型的对象（包括其他数组），然后产生一个新的含有传入数据的NumPy数组。以一个列表的转换为例：

```
In [19]: data1 = [6, 7.5, 8, 0, 1]

In [20]: arr1 = np.array(data1)

In [21]: arr1
Out[21]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

嵌套序列（比如由一组等长列表组成的列表）将会被转换为一个多维数组：

```
In [22]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [23]: arr2 = np.array(data2)

In [24]: arr2
Out[24]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

因为data2是列表的列表，NumPy数组arr2的两个维度的shape是从data2引入的。可以用属性ndim和shape验证：

```
In [25]: arr2.ndim
Out[25]: 2

In [26]: arr2.shape
Out[26]: (2, 4)
```

除非特别说明（稍后将会详细介绍），np.array会尝试为新建的这个数组推断出一个较为合适的数据类型。数据类型保存在一个特殊的dtype对象中。比如说，在上面的两个例子中，我们有：

```
In [27]: arr1.dtype
Out[27]: dtype('float64')

In [28]: arr2.dtype
Out[28]: dtype('int64')
```

除np.array之外，还有一些函数也可以新建数组。比如，zeros和ones分别可以创建指定长度或形状的全0或全1数组。empty可以创建一个没有任何具体值的数组。要用这些方法创建多维数组，只需传入一个表示形状的元组即可：

```
In [29]: np.zeros(10)
Out[29]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0

In [30]: np.zeros((3, 6))
Out[30]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])

In [31]: np.empty((2, 3, 2))
Out[31]:
array([[[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]],
       [[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]]])
```

> 注意：认为np.empty会返回全0数组的想法是不安全的。很多情况下（如前所示），它返回的都是一些未初始化的垃圾值。

arange是Python内置函数range的数组版：

```
In [32]: np.arange(15)
```

```
Out[32]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,
```

表4-1列出了一些数组创建函数。由于NumPy关注的是数值计算，因此，如果没有特别指定，数据类型基本都是float64（浮点数）。

函数	说明
array	将输入数据（列表、元组、数组或其它序列类型）转换为 ndarray。要么推断出 dtype，要么特别指定 dtype。默认直接复制输入数据
asarray	将输入转换为 ndarray，如果输入本身就是一个 ndarray 就不进行复制
arange	类似于内置的 range，但返回的是一个 ndarray 而不是列表
ones,ones_like	根据指定的形状和 dtype 创建一个全 1 数组。one_like 以另一个数组为参数，并根据其形状和 dtype 创建一个全 1 数组
zeros,zeros_like	类似于 ones 和 ones_like，只不过产生的是全 0 数组而已
empty,empty_like	创建新数组，只分配内存空间但不填充任何值
full,full_like	用 fill value 中的所有值，根据指定的形状和 dtype 创建一个数组。full_like 使用另一个数组，用相同的形状和 dtype 创建
eye,identity	创建一个正方的 N × N 单位矩阵（对角线为 1，其余为 0）

表4-1 数组创建函数

ndarray的数据类型

dtype（数据类型）是一个特殊的对象，它含有ndarray将一块内存解释为特定数据类型所需的信息：

```
In [33]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [34]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [35]: arr1.dtype
Out[35]: dtype('float64')

In [36]: arr2.dtype
Out[36]: dtype('int32')
```

dtype是NumPy灵活交互其它系统的源泉之一。多数情况下，它们直接映射到相应的机器表示，这使得“读写磁盘上的二进制数

据流”以及“集成低级语言代码（如C、Fortran）”等工作变得更加简单。数值型dtype的命名方式相同：一个类型名（如float或int），后面跟一个用于表示各元素位长的数字。标准的双精度浮点值（即Python中的float对象）需要占用8字节（即64位）。因此，该类型在NumPy中就记作float64。表4-2列出了NumPy所支持的全部数据类型。

> 笔记：记不住这些NumPy的dtype也没关系，新手更是如此。通常只需要知道你所处理的数据的大致类型是浮点数、复数、整数、布尔值、字符串，还是普通的Python对象即可。当你需要控制数据在内存和磁盘中的存储方式时（尤其是对大数据集），那就得了解如何控制存储类型。

表4-2：NumPy的数据类型

类型	类型代码	说明
int8、uint8	i1、u1	有符号和无符号的8位（1个字节）整型
int16、uint16	i2、u2	有符号和无符号的16位（2个字节）整型
int32、uint32	i4、u4	有符号和无符号的32位（4个字节）整型
int64、uint64	i8、u8	有符号和无符号的64位（8个字节）整型
float16	f2	半精度浮点数
float32	f4或f	标准的单精度浮点数。与C的float兼容
float64	f8或d	标准的双精度浮点数。与C的double和Python的float对象兼容
float128	f16或g	扩展精度浮点数
complex64、complex128、complex256	c8、c16、c32	分别用两个32位、64位或128位浮点数表示的复数
bool	?	存储True和False值的布尔类型

表4-2：NumPy的数据类型（续）

类型	类型代码	说明
object	O	Python对象类型
string_	S	固定长度的字符串类型（每个字符1个字节）。例如，要创建一个长度为10的字符串，应使用S10
unicode_	U	固定长度的unicode类型（字节数由平台决定）。跟字符串的定义方式一样（如U10）

你可以通过ndarray的astype方法明确地将一个数组从一个dtype转换成另一个dtype：

```
In [37]: arr = np.array([1, 2, 3, 4, 5])

In [38]: arr.dtype
Out[38]: dtype('int64')

In [39]: float_arr = arr.astype(np.float64)

In [40]: float_arr.dtype
Out[40]: dtype('float64')
```

在本例中，整数被转换成了浮点数。如果将浮点数转换成整数，则小数部分将会被截取删除：

```
In [41]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 1

In [42]: arr
Out[42]: array([  3.7,  -1.2,  -2.6,   0.5,  12.9,  1

In [43]: arr.astype(np.int32)
Out[43]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

如果某字符串数组表示的全是数字，也可以用astype将其转换为数值形式：

```
In [44]: numeric_strings = np.array(['1.25', '-9.6',

In [45]: numeric_strings.astype(float)
Out[45]: array([  1.25,  -9.6 ,  42.  ])
```

> 注意：使用numpy.string_类型时，一定要小心，因为NumPy的字符串数据是大小固定的，发生截取时，不会发出警告。pandas提供了更多非数值数据的便利的处理方法。

如果转换过程因为某种原因而失败了（比如某个不能被转换为float64的字符串），就会引发一个ValueError。这里，我比较懒，写的是float而不是np.float64；NumPy很聪明，它会将Python类型映射到等价的dtype上。

数组的dtype还有另一个属性：

```
In [46]: int_array = np.arange(10)

In [47]: calibers = np.array([.22, .270, .357, .380,

In [48]: int_array.astype(calibers.dtype)
Out[48]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7
```

你还可以用简洁的类型代码来表示dtype：

```
In [49]: empty_uint32 = np.empty(8, dtype='u4')

In [50]: empty_uint32
Out[50]:
array([         0, 1075314688,          0, 1075707904
       1075838976,          0, 1072693248], dtype=uin
```

> 笔记：调用astype总会创建一个新的数组（一个数据的备份），即使新的dtype与旧的dtype相同。

NumPy数组的运算

数组很重要，因为它使你不用编写循环即可对数据执行批量运算。NumPy用户称其为矢量化（vectorization）。大小相等的数组之间的任何算术运算都会将运算应用到元素级：

```
In [51]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [52]: arr
Out[52]:
```

```
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [53]: arr * arr
Out[53]:
array([[  1.,   4.,   9.],
       [ 16.,  25.,  36.]])

In [54]: arr - arr
Out[54]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

数组与标量的算术运算会将标量值传播到各个元素：

```
In [55]: 1 / arr
Out[55]:
array([[ 1.    ,  0.5   ,  0.3333],
       [ 0.25  ,  0.2   ,  0.1667]])

In [56]: arr ** 0.5
Out[56]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```

大小相同的数组之间的比较会生成布尔值数组：

```
In [57]: arr2 = np.array([[0., 4., 1.], [7., 2., 12.]

In [58]: arr2
Out[58]:
array([[  0.,   4.,   1.],
       [  7.,   2.,  12.]])

In [59]: arr2 > arr
Out[59]:
```

```
array([[False,  True, False],
       [ True, False,  True]], dtype=bool)
```

不同大小的数组之间的运算叫做广播（broadcasting），将在附录A中对其进行详细讨论。本书的内容不需要对广播机制有多深的理解。

基本的索引和切片

NumPy数组的索引是一个内容丰富的主题，因为选取数据子集或单个元素的方式有很多。一维数组很简单。从表面上看，它们跟Python列表的功能差不多：

```
In [60]: arr = np.arange(10)

In [61]: arr
Out[61]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [62]: arr[5]
Out[62]: 5

In [63]: arr[5:8]
Out[63]: array([5, 6, 7])

In [64]: arr[5:8] = 12

In [65]: arr
Out[65]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  
```

如上所示，当你将一个标量值赋值给一个切片时（如arr[5:8]=12），该值会自动传播（也就说后面将会讲到的“广播”）到整个选区。跟列表最重要的区别在于，数组切片是原始数组的视图。这意味着数据不会被复制，视图上的任何修改都会直接反映到源数组上。

作为例子，先创建一个arr的切片：

```
In [66]: arr_slice = arr[5:8]

In [67]: arr_slice
Out[67]: array([12, 12, 12])
```

现在，当我修稿arr_slice中的值，变动也会体现在原始数组arr中：

```
In [68]: arr_slice[1] = 12345

In [69]: arr
Out[69]: array([     0,      1,      2,      3,      4,
  9])
```

切片[:]会给数组中的所有值赋值：

```
In [70]: arr_slice[:] = 64

In [71]: arr
Out[71]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,
```

如果你刚开始接触NumPy，可能会对此感到惊讶（尤其是当你曾经用过其他热衷于复制数组数据的编程语言）。由于NumPy的设计目的是处理大数据，所以你可以想象一下，假如NumPy坚持要将数据复制来复制去的话会产生何等的性能和内存问题。

> 注意：如果你想要得到的是ndarray切片的一份副本而非视图，就需要明确地进行复制操作，例如`arr[5:8].copy()`。

对于高维度数组，能做的事情更多。在一个二维数组中，各索引位置上的元素不再是标量而是一维数组：

```
In [72]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7,
```

```
In [73]: arr2d[2]
Out[73]: array([7, 8, 9])
```

因此，可以对各个元素进行递归访问，但这样需要做的事情有点多。你可以传入一个以逗号隔开的索引列表来选取单个元素。也就是说，下面两种方式是等价的：

```
In [74]: arr2d[0][2]
Out[74]: 3

In [75]: arr2d[0, 2]
Out[75]: 3
```

图4-1说明了二维数组的索引方式。轴0作为行，轴1作为列。

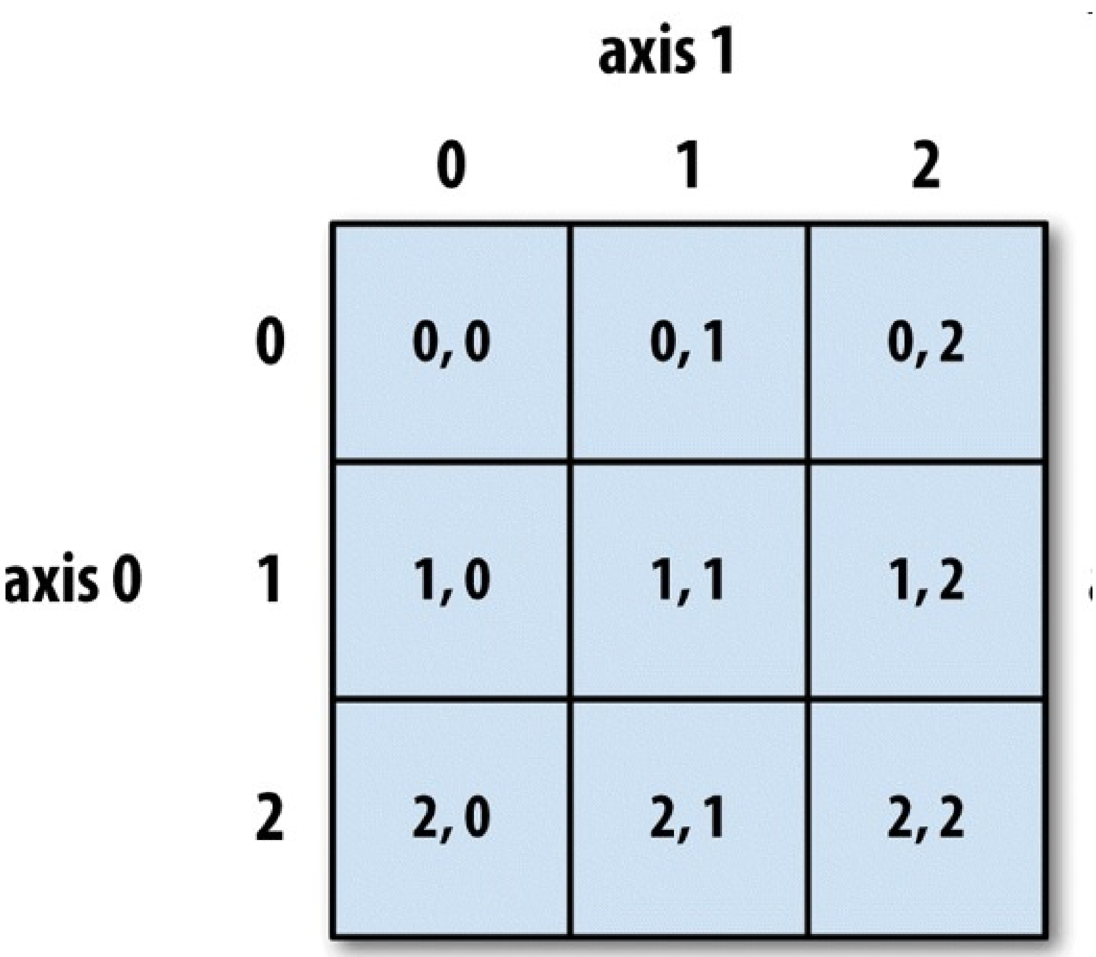


图4-1 NumPy数组中的元素索引

在多维数组中，如果省略了后面的索引，则返回对象会是一个维度低一点的ndarray（它含有高一级维度上的所有数据）。因此，在2×2×3数组arr3d中：

```
In [76]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[

In [77]: arr3d
Out[77]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

arr3d[0]是一个2×3数组：

```
In [78]: arr3d[0]
Out[78]:
array([[1, 2, 3],
       [4, 5, 6]])
```

标量值和数组都可以被赋值给arr3d[0]：

```
In [79]: old_values = arr3d[0].copy()

In [80]: arr3d[0] = 42

In [81]: arr3d
Out[81]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [82]: arr3d[0] = old_values
```

```
In [83]: arr3d
Out[83]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

相似的，arr3d[1,0]可以访问索引以(1,0)开头的那些值（以一维数组的形式返回）：

```
In [84]: arr3d[1, 0]
Out[84]: array([7, 8, 9])
```

虽然是用两步进行索引的，表达式是相同的：

```
In [85]: x = arr3d[1]

In [86]: x
Out[86]:
array([[ 7,  8,  9],
       [10, 11, 12]])

In [87]: x[0]
Out[87]: array([7, 8, 9])
```

注意，在上面所有这些选取数组子集的例子中，返回的数组都是视图。

切片索引

ndarray的切片语法跟Python列表这样的一维对象差不多：

```
In [88]: arr
Out[88]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  

In [89]: arr[1:6]
```

```
Out[89]: array([ 1,  2,  3,  4, 64])
```

对于之前的二维数组arr2d，其切片方式稍显不同：

```
In [90]: arr2d
Out[90]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [91]: arr2d[:2]
Out[91]:
array([[1, 2, 3],
       [4, 5, 6]])
```

可以看出，它是沿着第0轴（即第一个轴）切片的。也就是说，切片是沿着一个轴向选取元素的。表达式arr2d[:2]可以被认为是“选取arr2d的前两行”。

你可以一次传入多个切片，就像传入多个索引那样：

```
In [92]: arr2d[:2, 1:]
Out[92]:
array([[2, 3],
       [5, 6]])
```

像这样进行切片时，只能得到相同维数的数组视图。通过将整数索引和切片混合，可以得到低维度的切片。

例如，我可以选取第二行的前两列：

```
In [93]: arr2d[1, :2]
Out[93]: array([4, 5])
```

相似的，还可以选择第三列的前两行：

```
In [94]: arr2d[:2, 2]
```

```
Out[94]: array([3, 6])
```

图4-2对此进行了说明。注意，“只有冒号”表示选取整个轴，因此你可以像下面这样只对高维轴进行切片：

```
In [95]: arr2d[:, :1]
Out[95]:
array([[1],
       [4],
       [7]])
```

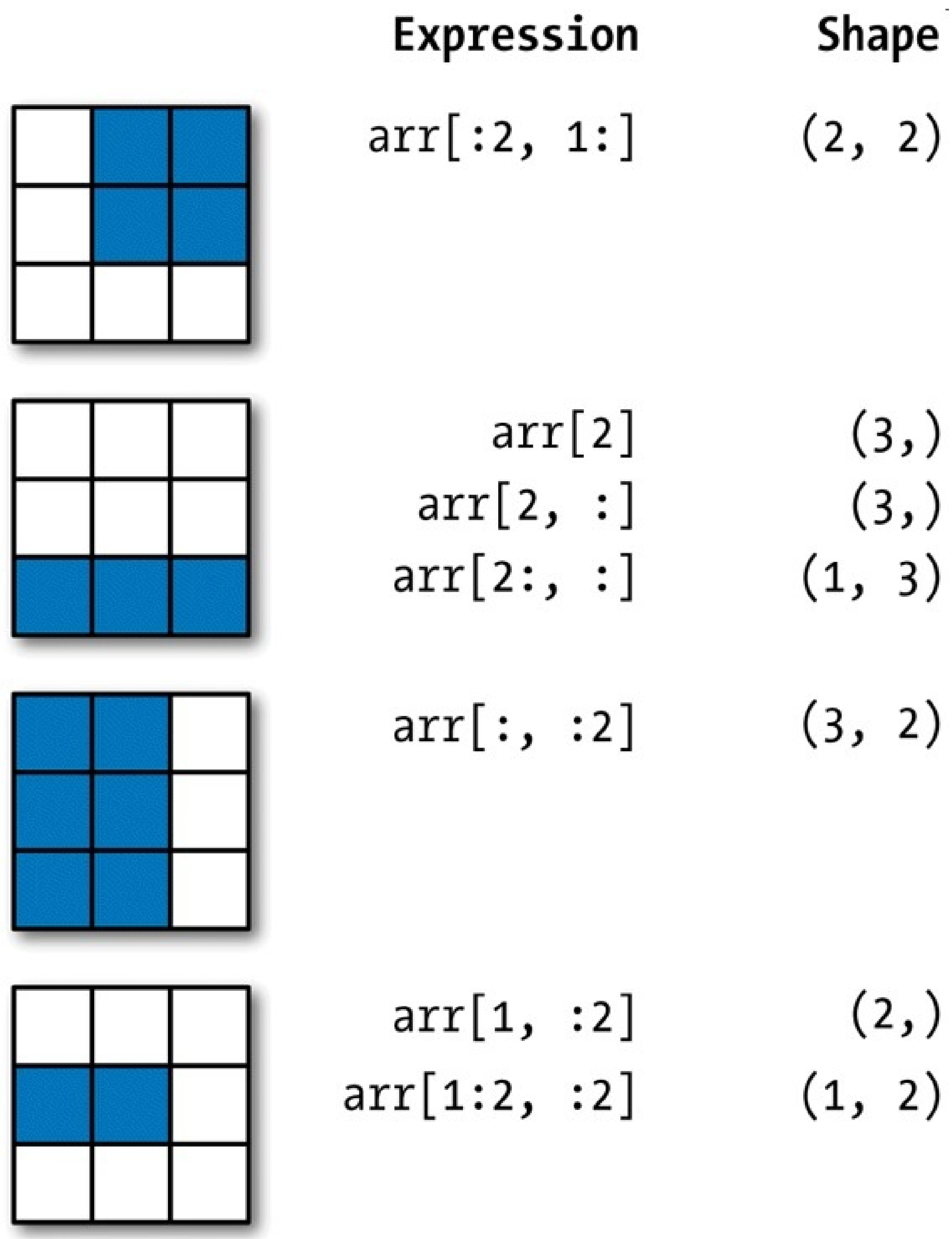


图4-2 二维数组切片

自然，对切片表达式的赋值操作也会被扩散到整个选区：

```
In [96]: arr2d[:2, 1:] = 0
```

```
In [97]: arr2d
Out[97]:
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

布尔型索引

来看这样一个例子，假设我们有一个用于存储数据的数组以及一个存储姓名的数组（含有重复项）。在这里，我将使用numpy.random中的randn函数生成一些正态分布的随机数据：

```
In [98]: names = np.array(['Bob', 'Joe', 'Will', 'Bob

In [99]: data = np.random.randn(7, 4)

In [100]: names
Out[100]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'J
      dtype='<U4')

In [101]: data
Out[101]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

假设每个名字都对应data数组中的一行，而我们想要选出对应于名字"Bob"的所有行。跟算术运算一样，数组的比较运算（如==）也是矢量化的。因此，对names和字符串"Bob"的比较运算将会产生一个布尔型数组：

```
In [102]: names == 'Bob'
Out[102]: array([ True, False, False,  True, False, Fa
```

这个布尔型数组可用于数组索引：

```
In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

布尔型数组的长度必须跟被索引的轴长度一致。此外，还可以将布尔型数组跟切片、整数（或整数序列，稍后将对此进行详细讲解）混合使用：

```
In [103]: data[names == 'Bob']
Out[103]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

> 注意：如果布尔型数组的长度不对，布尔型选择就会出错，因此一定要小心。

下面的例子，我选取了`names == 'Bob'`的行，并索引了列：

```
In [104]: data[names == 'Bob', 2:]
Out[104]:
array([[ 0.769 ,  1.2464],
       [-0.5397,  0.477 ]])

In [105]: data[names == 'Bob', 3]
Out[105]: array([ 1.2464,  0.477 ])
```

要选择除"Bob"以外的其他值，既可以使用不等于符号（!=），也可以通过~对条件进行否定：

```
In [106]: names != 'Bob'
```

```
Out[106]: array([False,  True,  True, False,  True,  
In [107]: data[~(names == 'Bob')]
Out[107]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

~操作符用来反转条件很好用：

```
In [108]: cond = names == 'Bob'

In [109]: data[~cond]
Out[109]:
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

选取这三个名字中的两个需要组合应用多个布尔条件，使用&（和）、|（或）之类的布尔算术运算符即可：

```
In [110]: mask = (names == 'Bob') | (names == 'Will')

In [111]: mask
Out[111]: array([ True, False,  True,  True,  True, F

In [112]: data[mask]
Out[112]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

通过布尔型索引选取数组中的数据，将总是创建数据的副本，即使返回一模一样的数组也是如此。

> 注意：Python关键字and和or在布尔型数组中无效。要是用&与|。

通过布尔型数组设置值是一种经常用到的手段。为了将data中的所有负值都设置为0，我们只需：

```
In [113]: data[data < 0] = 0

In [114]: data
Out[114]:
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072,  0.    ,  0.275 ,  0.2289],
       [ 1.3529,  0.8864,  0.    ,  0.    ],
       [ 1.669 ,  0.    ,  0.    ,  0.477 ],
       [ 3.2489,  0.    ,  0.    ,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0.    ,  0.    ,  0.    ,  0.    ]])
```

通过一维布尔数组设置整行或列的值也很简单：

```
In [115]: data[names != 'Joe'] = 7

In [116]: data
Out[116]:
array([[ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 1.0072,  0.    ,  0.275 ,  0.2289],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 7.    ,  7.    ,  7.    ,  7.    ],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [ 0.    ,  0.    ,  0.    ,  0.    ]])
```

后面会看到，这类二维数据的操作也可以用pandas方便的来

做。

花式索引

花式索引（Fancy indexing）是一个NumPy术语，它指的是利用整数数组进行索引。假设我们有一个8×4数组：

```
In [117]: arr = np.empty((8, 4))

In [118]: for i in range(8):
   .....:     arr[i] = i

In [119]: arr
Out[119]:
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.]])
```

为了以特定顺序选取行子集，只需传入一个用于指定顺序的整数列表或ndarray即可：

```
In [120]: arr[[4, 3, 0, 6]]
Out[120]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

这段代码确实达到我们的要求了！使用负数索引将会从末尾开始选取行：

```
In [121]: arr[[-3, -5, -7]]
```

```
Out[121]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```

一次传入多个索引数组会有一点特别。它返回的是一个一维数组，其中的元素对应各个索引元组：

```
In [122]: arr = np.arange(32).reshape((8, 4))

In [123]: arr
Out[123]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])

In [124]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[124]: array([ 4, 23, 29, 10])
```

附录A中会详细介绍reshape方法。

最终选出的是元素(1,0)、(5,3)、(7,1)和(2,2)。无论数组是多少维的，花式索引总是一维的。

这个花式索引的行为可能会跟某些用户的预期不一样（包括我在内），选取矩阵的行列子集应该是矩形区域的形式才对。下面是得到该结果的一个办法：

```
In [125]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[125]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
```

```
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

记住，花式索引跟切片不一样，它总是将数据复制到新数组中。

数组转置和轴对换

转置是重塑的一种特殊形式，它返回的是源数据的视图（不会进行任何复制操作）。数组不仅有transpose方法，还有一个特殊的T属性：

```
In [126]: arr = np.arange(15).reshape((3, 5))

In [127]: arr
Out[127]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

In [128]: arr.T
Out[128]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

在进行矩阵计算时，经常需要用到该操作，比如利用np.dot计算矩阵内积：

```
In [129]: arr = np.random.randn(6, 3)

In [130]: arr
Out[130]:
array([[-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329],
       [-2.3594, -0.1995, -1.542 ],
```

```
       [-0.9707, -1.307 ,  0.2863],
       [ 0.378 , -0.7539,  0.3313],
       [ 1.3497,  0.0699,  0.2467]])

In [131]: np.dot(arr.T, arr)
Out[131]:
array([[ 9.2291,  0.9394,  4.948 ],
       [ 0.9394,  3.7662, -1.3622],
       [ 4.948 , -1.3622,  4.3437]])
```

对于高维数组，transpose需要得到一个由轴编号组成的元组才能对这些轴进行转置（比较费脑子）：

```
In [132]: arr = np.arange(16).reshape((2, 2, 4))

In [133]: arr
Out[133]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [134]: arr.transpose((1, 0, 2))
Out[134]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

这里，第一个轴被换成了第二个，第二个轴被换成了第一个，最后一个轴不变。

简单的转置可以使用.T，它其实就是进行轴对换而已。ndarray还有一个swapaxes方法，它需要接受一对轴编号：

```
In [135]: arr
Out[135]:
```

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [136]: arr.swapaxes(1, 2)
Out[136]:
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],
       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

swapaxes也是返回源数据的视图（不会进行任何复制操作）。

4.2 通用函数：快速的元素级数组函数

通用函数（即ufunc）是一种对ndarray中的数据执行元素级运算的函数。你可以将其看做简单函数（接受一个或多个标量值，并产生一个或多个标量值）的矢量化包装器。

许多ufunc都是简单的元素级变体，如sqrt和exp：

```
In [137]: arr = np.arange(10)

In [138]: arr
Out[138]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [139]: np.sqrt(arr)
Out[139]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  
        2.6458,  2.8284,  3.    ])

In [140]: np.exp(arr)
```

```
Out[140]:
array([    1.    ,     2.7183,     7.3891,    20.0855
         148.4132,   403.4288,  1096.6332,  2980.958
```

这些都是一元（unary）ufunc。另外一些（如add或maximum）接受2个数组（因此也叫二元（binary）ufunc），并返回一个结果数组：

```
In [141]: x = np.random.randn(8)

In [142]: y = np.random.randn(8)

In [143]: x
Out[143]:
array([-0.0119,  1.0048,  1.3272, -0.9193, -1.5491,  
       -0.6605])

In [144]: y
Out[144]:
array([ 0.8626, -0.01  ,  0.05  ,  0.6702,  0.853 , -
       -2.3042])

In [145]: np.maximum(x, y)
Out[145]:
array([ 0.8626,  1.0048,  1.3272,  0.6702,  0.853 ,  
       -0.6605])
```

这里，numpy.maximum计算了x和y中元素级别最大的元素。

虽然并不常见，但有些ufunc的确可以返回多个数组。modf就是一个例子，它是Python内置函数divmod的矢量化版本，它会返回浮点数数组的小数和整数部分：

```
In [146]: arr = np.random.randn(7) * 5

In [147]: arr
```

```
Out[147]: array([-3.2623, -6.0915, -6.663 ,  5.3731,

In [148]: remainder, whole_part = np.modf(arr)

In [149]: remainder
Out[149]: array([-0.2623, -0.0915, -0.663 ,  0.3731,
0.6182,  0.45  ,  0.0077])

In [150]: whole_part
Out[150]: array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

Ufuncs接受out选项参数，可以让它们在数组的原地进行操作：

```
In [151]: arr
Out[151]: array([-3.2623, -6.0915, -6.663 ,  5.3731,

In [152]: np.sqrt(arr)
Out[152]: array([    nan,     nan,     nan,  2.318 ,

In [153]: np.sqrt(arr, arr)
Out[153]: array([    nan,     nan,     nan,  2.318 ,

In [154]: arr
Out[154]: array([    nan,     nan,     nan,  2.318 ,
```

表4-3和表4-4分别列出了一些一元和二元ufunc。

表4-3：一元ufunc

函数	说明
abs、fabs	计算整数、浮点数或复数的绝对值。对于非复数值，可以使用更快的fabs
sqrt	计算各元素的平方根。相当于arr ** 0.5
square	计算各元素的平方。相当于arr ** 2
exp	计算各元素的指数e^x
log、log10、log2、log1p	分别为自然对数（底数为e）、底数为10的log、底数为2的log、log(1 + x)
sign	计算各元素的正负号：1（正数）、0（零）、－1（负数）
ceil	计算各元素的ceiling值，即大于等于该值的最小整数
floor	计算各元素的floor值，即小于等于该值的最大整数
rint	将各元素值四舍五入到最接近的整数，保留dtype
modf	将数组的小数和整数部分以两个独立数组的形式返回
isnan	返回一个表示“哪些值是NaN（这不是一个数字）”的布尔型数组
isfinite、isinf	分别返回一个表示“哪些元素是有穷的（非inf，非NaN）”或“哪些元素是无穷的”的布尔型数组
cos、cosh、sin、sinh、tan、tanh	普通型和双曲型三角函数
arccos、arccosh、arcsin、arcsinh、arctan、arctanh	反三角函数
logical_not	计算各元素not x的真值。相当于-arr

表4-4：二元ufunc

函数	说明
add	将数组中对应的元素相加
subtract	从第一个数组中减去第二个数组中的元素
multiply	数组元素相乘
divide、floor_divide	除法或向下圆整除法（丢弃余数）
power	对第一个数组中的元素A，根据第二个数组中的相应元素B，计算A^B
maximum、fmax	元素级的最大值计算。fmax将忽略NaN
minimum、fmin	元素级的最小值计算。fmin将忽略NaN
mod	元素级的求模计算（除法的余数）
copysign	将第二个数组中的值的符号复制给第一个数组中的值
greater、greater_equal、less、less_equal、equal、not_equal	执行元素级的比较运算，最终产生布尔型数组。相当于中缀运算符>、>=、<、<=、==、!=
logical_and、logical_or、logical_xor	执行元素级的真值逻辑运算。相当于中缀运算符&、|、^

4.3 利用数组进行数据处理

NumPy数组使你可以将许多种数据处理任务表述为简洁的数组表达式（否则需要编写循环）。用数组表达式代替循环的做法，通常被称为矢量化。一般来说，矢量化数组运算要比等价的纯Python方式快上一两个数量级（甚至更多），尤其是各种数值计算。在后面内容中（见附录A）我将介绍广播，这是一种针对矢量化计算的强大手段。

作为简单的例子，假设我们想要在一组值（网格型）上计算函数`sqrt(x^2+y^2)`。np.meshgrid函数接受两个一维数组，并产生两个二维矩阵（对应于两个数组中所有的(x,y)对）：

```
In [155]: points = np.arange(-5, 5, 0.01) # 1000 equa

In [156]: xs, ys = np.meshgrid(points, points)
In [157]: ys
Out[157]:
```

```
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99
```

现在，对该函数的求值运算就好办了，把这两个数组当做两个浮点数那样编写表达式即可：

```
In [158]: z = np.sqrt(xs ** 2 + ys ** 2)

In [159]: z
Out[159]:
array([[ 7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0
       ...,
       [ 7.0499,  7.0428,  7.0357, ...,  7.0286,  7.0
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0
```

作为第9章的先导，我用matplotlib创建了这个二维数组的可视化：

```
In [160]: import matplotlib.pyplot as plt

In [161]: plt.imshow(z, cmap=plt.cm.gray); plt.colorb
Out[161]: <matplotlib.colorbar.Colorbar at 0x7f715e3f

In [162]: plt.title("Image plot of $\sqrt{x^2 + y^2}$
Out[162]: <matplotlib.text.Text at 0x7f715d2de748>
```

见图4-3。这张图是用matplotlib的imshow函数创建的。

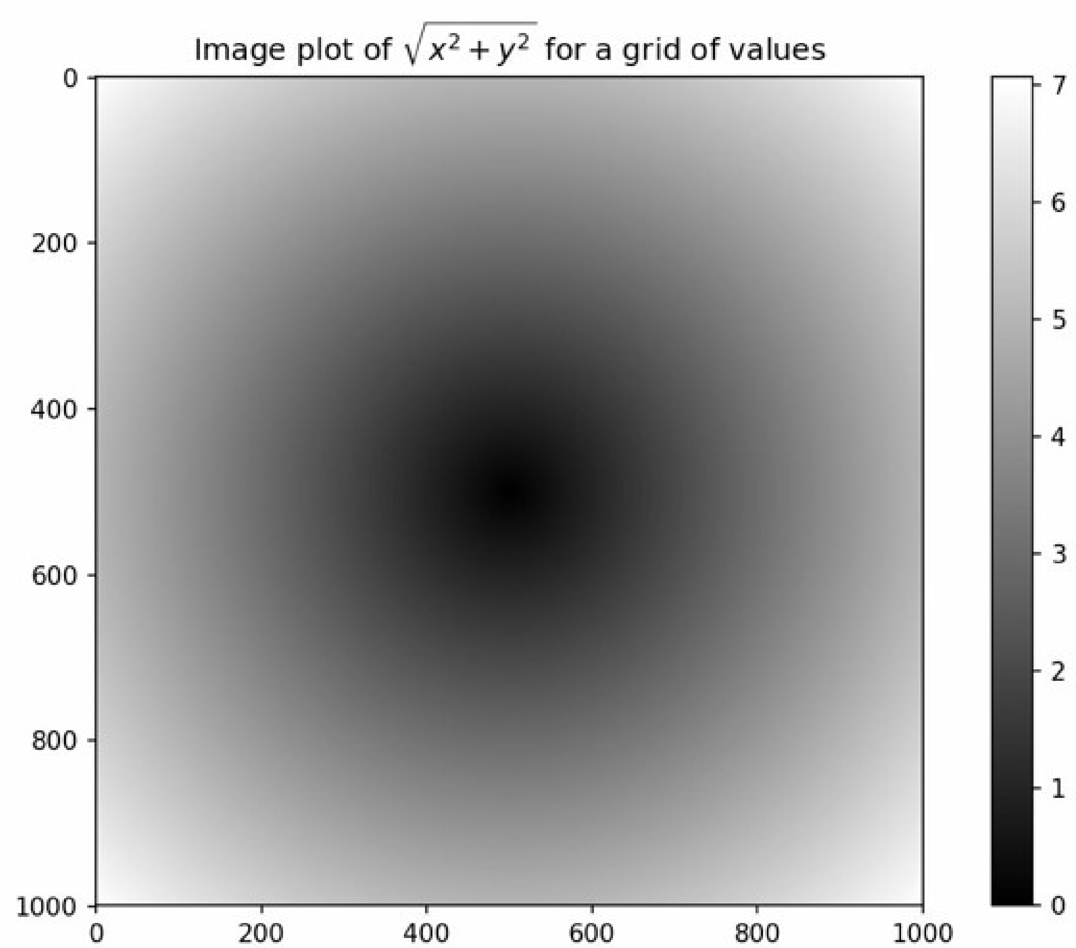


图4-3 根据网格对函数求值的结果

将条件逻辑表述为数组运算

numpy.where函数是三元表达式x if condition else y的矢量化版本。假设我们有一个布尔数组和两个值数组：

```
In [165]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [166]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [167]: cond = np.array([True, False, True, True, Fa
```

假设我们想要根据cond中的值选取xarr和yarr的值：当cond中的值为True时，选取xarr的值，否则从yarr中选取。列表推导式的

写法应该如下所示：

```
In [168]: result = [(x if c else y)
   .....:           for x, y, c in zip(xarr, yarr, co

In [169]: result
Out[169]: [1.1000000000000001, 2.2000000000000002, 1.
```

这有几个问题。第一，它对大数组的处理速度不是很快（因为所有工作都是由纯Python完成的）。第二，无法用于多维数组。若使用np.where，则可以将该功能写得非常简洁：

```
In [170]: result = np.where(cond, xarr, yarr)

In [171]: result
Out[171]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

np.where的第二个和第三个参数不必是数组，它们都可以是标量值。在数据分析工作中，where通常用于根据另一个数组而产生一个新的数组。假设有一个由随机数据组成的矩阵，你希望将所有正值替换为2，将所有负值替换为－2。若利用np.where，则会非常简单：

```
In [172]: arr = np.random.randn(4, 4)

In [173]: arr
Out[173]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 0.2229,  0.0513, -1.1577,  0.8167],
       [ 0.4336,  1.0107,  1.8249, -0.9975],
       [ 0.8506, -0.1316,  0.9124,  0.1882]])

In [174]: arr > 0
Out[174]:
array([[False, False, False, False],
       [ True,  True, False,  True],
```

```
       [ True,  True,  True, False],
       [ True, False,  True,  True]], dtype=bool)

In [175]: np.where(arr > 0, 2, -2)
Out[175]:
array([[-2, -2, -2, -2],
       [ 2,  2, -2,  2],
       [ 2,  2,  2, -2],
       [ 2, -2,  2,  2]])
```

使用np.where，可以将标量和数组结合起来。例如，我可用常数2替换arr中所有正的值：

```
In [176]: np.where(arr > 0, 2, arr) # set only positi
Out[176]:
array([[-0.5031, -0.6223, -0.9212, -0.7262],
       [ 2.    ,  2.    , -1.1577,  2.    ],
       [ 2.    ,  2.    ,  2.    , -0.9975],
       [ 2.    , -0.1316,  2.    ,  2.    ]])
```

传递给where的数组大小可以不相等，甚至可以是标量值。

数学和统计方法

可以通过数组上的一组数学函数对整个数组或某个轴向的数据进行统计计算。sum、mean以及标准差std等聚合计算（aggregation，通常叫做约简（reduction））既可以当做数组的实例方法调用，也可以当做顶级NumPy函数使用。

这里，我生成了一些正态分布随机数据，然后做了聚类统计：

```
In [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
```

```
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])

In [179]: arr.mean()
Out[179]: 0.19607051119998253

In [180]: np.mean(arr)
Out[180]: 0.19607051119998253

In [181]: arr.sum()
Out[181]: 3.9214102239996507
```

mean和sum这类的函数可以接受一个axis选项参数，用于计算该轴向上的统计值，最终结果是一个少一维的数组：

```
In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

这里，arr.mean(1)是“计算行的平均值”，arr.sum(0)是“计算每列的和”。

其他如cumsum和cumprod之类的方法则不聚合，而是产生一个由中间结果组成的数组：

```
In [184]: arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])

In [185]: arr.cumsum()
Out[185]: array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

在多维数组中，累加函数（如cumsum）返回的是同样大小的数组，但是会根据每个低维的切片沿着标记轴计算部分聚类：

```
In [186]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7

In [187]: arr
Out[187]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [188]: arr.cumsum(axis=0)
Out[188]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [189]: arr.cumprod(axis=1)
Out[189]:
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

表4-5列出了全部的基本数组统计方法。后续章节中有很多例子都会用到这些方法。

表4-5：基本数组统计方法

方法	说明
sum	对数组中全部或某轴向的元素求和。零长度的数组的sum为0
mean	算术平均数。零长度的数组的mean为NaN
std、var	分别为标准差和方差，自由度可调（默认为n）
min、max	最大值和最小值
argmin、argmax	分别为最大和最小元素的索引
cumsum	所有元素的累计和
cumprod	所有元素的累计积

用于布尔型数组的方法

在上面这些方法中，布尔值会被强制转换为1（True）和0（False）。因此，sum经常被用来对布尔型数组中的True值计数：

```
In [190]: arr = np.random.randn(100)

In [191]: (arr > 0).sum() # Number of positive values
Out[191]: 42
```

另外还有两个方法any和all，它们对布尔型数组非常有用。any用于测试数组中是否存在一个或多个True，而all则检查数组中所有值是否都是True：

```
In [192]: bools = np.array([False, False, True, False

In [193]: bools.any()
Out[193]: True

In [194]: bools.all()
Out[194]: False
```

这两个方法也能用于非布尔型数组，所有非0元素将会被当做True。

排序

跟Python内置的列表类型一样，NumPy数组也可以通过sort方法就地排序：

```
In [195]: arr = np.random.randn(6)

In [196]: arr
Out[196]: array([ 0.6095, -0.4938,  1.24  , -0.1357,
```

```
In [197]: arr.sort()

In [198]: arr
Out[198]: array([-0.8469, -0.4938, -0.1357,  0.6095,
```

多维数组可以在任何一个轴向上进行排序，只需将轴编号传给sort即可：

```
In [199]: arr = np.random.randn(5, 3)

In [200]: arr
Out[200]:
array([[ 0.6033,  1.2636, -0.2555],
       [-0.4457,  0.4684, -0.9616],
       [-1.8245,  0.6254,  1.0229],
       [ 1.1074,  0.0909, -0.3501],
       [ 0.218 , -0.8948, -1.7415]])

In [201]: arr.sort(1)

In [202]: arr
Out[202]:
array([[-0.2555,  0.6033,  1.2636],
       [-0.9616, -0.4457,  0.4684],
       [-1.8245,  0.6254,  1.0229],
       [-0.3501,  0.0909,  1.1074],
       [-1.7415, -0.8948,  0.218 ]])
```

顶级方法np.sort返回的是数组的已排序副本，而就地排序则会修改数组本身。计算数组分位数最简单的办法是对其进行排序，然后选取特定位置的值：

```
In [203]: large_arr = np.random.randn(1000)

In [204]: large_arr.sort()
```

```
In [205]: large_arr[int(0.05 * len(large_arr))] # 5% (
Out[205]: -1.5311513550102103
```

更多关于NumPy排序方法以及诸如间接排序之类的高级技术，请参阅附录A。在pandas中还可以找到一些其他跟排序有关的数据操作（比如根据一列或多列对表格型数据进行排序）。

唯一化以及其它的集合逻辑

NumPy提供了一些针对一维ndarray的基本集合运算。最常用的可能要数np.unique了，它用于找出数组中的唯一值并返回已排序的结果：

```
In [206]: names = np.array(['Bob', 'Joe', 'Will', 'Bo

In [207]: np.unique(names)
Out[207]:
array(['Bob', 'Joe', 'Will'],
      dtype='<U4')

In [208]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4]

In [209]: np.unique(ints)
Out[209]: array([1, 2, 3, 4])
```

拿跟np.unique等价的纯Python代码来对比一下：

```
In [210]: sorted(set(names))
Out[210]: ['Bob', 'Joe', 'Will']
```

另一个函数np.in1d用于测试一个数组中的值在另一个数组中的成员资格，返回一个布尔型数组：

```
In [211]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [212]: np.in1d(values, [2, 3, 6])
```

```
Out[212]: array([ True, False, False,  True,  True, Fa
```

NumPy中的集合函数请参见表4-6。

表4-6：数组的集合运算

方法	说明
unique(x)	计算x中的唯一元素，并返回有序结果
intersect1d(x, y)	计算x和y中的公共元素，并返回有序结果
union1d(x, y)	计算x和y的并集，并返回有序结果
in1d(x, y)	得到一个表示“x的元素是否包含于y”的布尔型数组
setdiff1d(x, y)	集合的差，即元素在x中且不在y中
setxor1d(x, y)	集合的对称差，即存在于一个数组中但不同时存在于两个数组中的元素[译注2]

4.4 用于数组的文件输入输出

NumPy能够读写磁盘上的文本数据或二进制数据。这一小节只讨论NumPy的内置二进制格式，因为更多的用户会使用pandas或其它工具加载文本或表格数据（见第6章）。

np.save和np.load是读写磁盘数组数据的两个主要函数。默认情况下，数组是以未压缩的原始二进制格式保存在扩展名为.npy的文件中的：

```
In [213]: arr = np.arange(10)

In [214]: np.save('some_array', arr)
```

如果文件路径末尾没有扩展名.npy，则该扩展名会被自动加上。然后就可以通过np.load读取磁盘上的数组：

```
In [215]: np.load('some_array.npy')
Out[215]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

通过np.savez可以将多个数组保存到一个未压缩文件中，将数组

以关键字参数的形式传入即可：

```
In [216]: np.savez('array_archive.npz', a=arr, b=arr)
```

加载.npz文件时，你会得到一个类似字典的对象，该对象会对各个数组进行延迟加载：

```
In [217]: arch = np.load('array_archive.npz')

In [218]: arch['b']
Out[218]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

如果数据压缩的很好，就可以使用numpy.savez_compressed：

```
In [219]: np.savez_compressed('arrays_compressed.npz'
```

4.5 线性代数

线性代数（如矩阵乘法、矩阵分解、行列式以及其他方阵数学等）是任何数组库的重要组成部分。不像某些语言（如MATLAB），通过*对两个二维数组相乘得到的是一个元素级的积，而不是一个矩阵点积。因此，NumPy提供了一个用于矩阵乘法的dot函数（既是一个数组方法也是numpy命名空间中的一个函数）：

```
In [223]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [224]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [225]: x
Out[225]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [226]: y
```

```
Out[226]:
array([[  6.,  23.],
       [ -1.,   7.],
       [  8.,   9.]])

In [227]: x.dot(y)
Out[227]:
array([[  28.,   64.],
       [  67.,  181.]])
```

x.dot(y)等价于np.dot(x, y)：

```
In [228]: np.dot(x, y)
Out[228]:
array([[  28.,   64.],
       [  67.,  181.]])
```

一个二维数组跟一个大小合适的一维数组的矩阵点积运算之后将会得到一个一维数组：

```
In [229]: np.dot(x, np.ones(3))
Out[229]: array([  6.,  15.])
```

@符（类似Python 3.5）也可以用作中缀运算符，进行矩阵乘法：

```
In [230]: x @ np.ones(3)
Out[230]: array([  6.,  15.])
```

numpy.linalg中有一组标准的矩阵分解运算以及诸如求逆和行列式之类的东西。它们跟MATLAB和R等语言所使用的是相同的行业标准线性代数库，如BLAS、LAPACK、Intel MKL（Math Kernel Library，可能有，取决于你的NumPy版本）等：

```
In [231]: from numpy.linalg import inv, qr

In [232]: X = np.random.randn(5, 5)
```

```
In [233]: mat = X.T.dot(X)

In [234]: inv(mat)
Out[234]:
array([[  933.1189,   871.8258, -1417.6902, -1460.4005
       [  871.8258,   815.3929, -1325.9965, -1365.9242
       [-1417.6902, -1325.9965,  2158.4424,  2222.0191
       [-1460.4005, -1365.9242,  2222.0191,  2289.0575
       [ 1782.1391,  1666.9347, -2711.6822, -2793.422

In [235]: mat.dot(inv(mat))
Out[235]:
array([[ 1.,  0., -0., -0., -0.],
       [-0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [-0.,  0.,  0.,  1., -0.],
       [-0.,  0.,  0.,  0.,  1.]])

In [236]: q, r = qr(mat)

In [237]: r
Out[237]:
array([[-1.6914,  4.38  ,  0.1757,  0.4075, -0.7838],
       [ 0.    , -2.6436,  0.1939, -3.072 , -1.0702],
       [ 0.    ,  0.    , -0.8138,  1.5414,  0.6155],
       [ 0.    ,  0.    ,  0.    , -2.6445, -2.1669],
       [ 0.    ,  0.    ,  0.    ,  0.    ,  0.0002]]
```

表达式X.T.dot(X)计算X和它的转置X.T的点积。

表4-7中列出了一些最常用的线性代数函数。

表4-7：常用的numpy.linalg函数

函数	说明
diag	以一维数组的形式返回方阵的对角线（或非对角线）元素，或将一维数组转换为方阵（非对角线元素为0）
dot	矩阵乘法
trace	计算对角线元素的和
det	计算矩阵行列式
eig	计算方阵的本征值和本征向量
inv	计算方阵的逆
pinv	计算矩阵的Moore-Penrose伪逆
qr	计算QR分解
svd	计算奇异值分解（SVD）
solve	解线性方程组Ax = b，其中A为一个方阵
lstsq	计算Ax = b的最小二乘解

4.6 伪随机数生成

numpy.random模块对Python内置的random进行了补充，增加了一些用于高效生成多种概率分布的样本值的函数。例如，你可以用normal来得到一个标准正态分布的4×4样本数组：

```
In [238]: samples = np.random.normal(size=(4, 4))

In [239]: samples
Out[239]:
array([[ 0.5732,  0.1933,  0.4429,  1.2796],
       [ 0.575 ,  0.4339, -0.7658, -1.237 ],
       [-0.5367,  1.8545, -0.92  , -0.1082],
       [ 0.1525,  0.9435, -1.0953, -0.144 ]])
```

而Python内置的random模块则只能一次生成一个样本值。从下面的测试结果中可以看出，如果需要产生大量样本值，numpy.random快了不止一个数量级：

```
In [240]: from random import normalvariate
```

```
In [241]: N = 1000000

In [242]: %timeit samples = [normalvariate(0, 1) for _
1.77 s +- 126 ms per loop (mean +- std. dev. of 7 run

In [243]: %timeit np.random.normal(size=N)
61.7 ms +- 1.32 ms per loop (mean +- std. dev. of 7 r
```

我们说这些都是伪随机数，是因为它们都是通过算法基于随机数生成器种子，在确定性的条件下生成的。你可以用NumPy的np.random.seed更改随机数生成种子：

```
In [244]: np.random.seed(1234)
```

numpy.random的数据生成函数使用了全局的随机种子。要避免全局状态，你可以使用numpy.random.RandomState，创建一个与其它隔离的随机数生成器：

```
In [245]: rng = np.random.RandomState(1234)

In [246]: rng.randn(10)
Out[246]:
array([ 0.4714, -1.191 ,  1.4327, -0.3127, -0.7206,  (
       -0.6365,  0.0157, -2.2427])
```

表4-8列出了numpy.random中的部分函数。在下一节中，我将给出一些利用这些函数一次性生成大量样本值的范例。

表4-8：部分numpy.random函数

函数	说明
seed	确定随机数生成器的种子
permutation	返回一个序列的随机排列或返回一个随机排列的范围
shuffle	对一个序列就地随机排列
rand	产生均匀分布的样本值
randint	从给定的上下限范围内随机选取整数
randn	产生正态分布（平均值为0，标准差为1）的样本值，类似于MATLAB接口
binomial	产生二项分布的样本值
normal	产生正态（高斯）分布的样本值
beta	产生Beta分布的样本值
chisquare	产生卡方分布的样本值
gamma	产生Gamma分布的样本值
uniform	产生在[0, 1)中均匀分布的样本值

4.7 示例：随机漫步

我们通过模拟随机漫步来说明如何运用数组运算。先来看一个简单的随机漫步的例子：从0开始，步长1和－1出现的概率相等。

下面是一个通过内置的random模块以纯Python的方式实现1000步的随机漫步：

```
In [247]: import random
   .....: position = 0
   .....: walk = [position]
   .....: steps = 1000
   .....: for i in range(steps):
   .....:     step = 1 if random.randint(0, 1) else -
   .....:     position += step
   .....:     walk.append(position)
   .....:
```

图4-4是根据前100个随机漫步值生成的折线图：

```
In [249]: plt.plot(walk[:100])
```

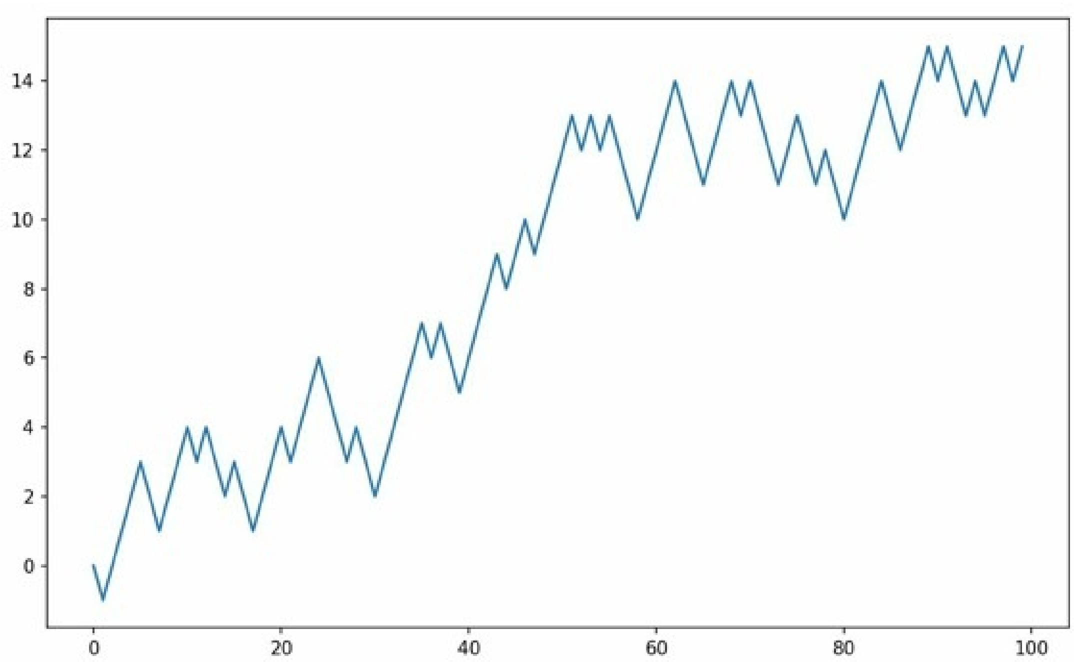


图4-4 简单的随机漫步

不难看出，这其实就是随机漫步中各步的累计和，可以用一个数组运算来实现。因此，我用np.random模块一次性随机产生1000个“掷硬币”结果（即两个数中任选一个），将其分别设置为1或－1，然后计算累计和：

```
In [251]: nsteps = 1000

In [252]: draws = np.random.randint(0, 2, size=nsteps

In [253]: steps = np.where(draws > 0, 1, -1)

In [254]: walk = steps.cumsum()
```

有了这些数据之后，我们就可以沿着漫步路径做一些统计工作了，比如求取最大值和最小值：

```
In [255]: walk.min()
Out[255]: -3

In [256]: walk.max()
Out[256]: 31
```

现在来看一个复杂点的统计任务——首次穿越时间，即随机漫步过程中第一次到达某个特定值的时间。假设我们想要知道本次随机漫步需要多久才能距离初始0点至少10步远（任一方向均可）。np.abs(walk)>=10可以得到一个布尔型数组，它表示的是距离是否达到或超过10，而我们想要知道的是第一个10或－10的索引。可以用argmax来解决这个问题，它返回的是该布尔型数组第一个最大值的索引（True就是最大值）：

```
In [257]: (np.abs(walk) >= 10).argmax()
Out[257]: 37
```

注意，这里使用argmax并不是很高效，因为它无论如何都会对数组进行完全扫描。在本例中，只要发现了一个True，那我们就知道它是个最大值了。

一次模拟多个随机漫步

如果你希望模拟多个随机漫步过程（比如5000个），只需对上面的代码做一点点修改即可生成所有的随机漫步过程。只要给numpy.random的函数传入一个二元元组就可以产生一个二维数组，然后我们就可以一次性计算5000个随机漫步过程（一行一个）的累计和了：

```
In [258]: nwalks = 5000

In [259]: nsteps = 1000

In [260]: draws = np.random.randint(0, 2, size=(nwalk

In [261]: steps = np.where(draws > 0, 1, -1)
```

```
In [262]: walks = steps.cumsum(1)

In [263]: walks
Out[263]:
array([[  1,   0,   1, ...,   8,   7,   8],
       [  1,   0,  -1, ...,  34,  33,  32],
       [  1,   0,  -1, ...,   4,   5,   4],
       ...,
       [  1,   2,   1, ...,  24,  25,  26],
       [  1,   2,   3, ...,  14,  13,  14],
       [ -1,  -2,  -3, ..., -24, -23, -22]])
```

现在，我们来计算所有随机漫步过程的最大值和最小值：

```
In [264]: walks.max()
Out[264]: 138

In [265]: walks.min()
Out[265]: -133
```

得到这些数据之后，我们来计算30或－30的最小穿越时间。这里稍微复杂些，因为不是5000个过程都到达了30。我们可以用any方法来对此进行检查：

```
In [266]: hits30 = (np.abs(walks) >= 30).any(1)

In [267]: hits30
Out[267]: array([False,  True, False, ..., False,  Tr

In [268]: hits30.sum() # Number that hit 30 or -30
Out[268]: 3410
```

然后我们利用这个布尔型数组选出那些穿越了30（绝对值）的随机漫步（行），并调用argmax在轴1上获取穿越时间：

```
In [269]: crossing_times = (np.abs(walks[hits30]) >= 3

In [270]: crossing_times.mean()
Out[270]: 498.88973607038122
```

请尝试用其他分布方式得到漫步数据。只需使用不同的随机数生成函数即可，如normal用于生成指定均值和标准差的正态分布数据：

```
In [271]: steps = np.random.normal(loc=0, scale=0.25,
   .....:                          size=(nwalks, nste
```

4.8 结论

虽然本书剩下的章节大部分是用pandas规整数据，我们还是会用到相似的基于数组的计算。在附录A中，我们会深入挖掘NumPy的特点，进一步学习数组的技巧。

第5章 pandas入门

pandas是本书后续内容的首选库。它含有使数据清洗和分析工作变得更快更简单的数据结构和操作工具。pandas经常和其它工具一同使用，如数值计算工具NumPy和SciPy，分析库statsmodels和scikit-learn，和数据可视化库matplotlib。pandas是基于NumPy数组构建的，特别是基于数组的函数和不使用for循环的数据处理。

虽然pandas采用了大量的NumPy编码风格，但二者最大的不同是pandas是专门为处理表格和混杂数据设计的。而NumPy更适合处理统一的数值数组数据。

自从2010年pandas开源以来，pandas逐渐成长为一个非常大的库，应用于许多真实案例。开发者社区已经有了800个独立的贡献者，他们在解决日常数据问题的同时为这个项目提供贡献。

在本书后续部分中，我将使用下面这样的pandas引入约定：

```
In [1]: import pandas as pd
```

因此，只要你在代码中看到pd.，就得想到这是pandas。因为Series和DataFrame用的次数非常多，所以将其引入本地命名空间中会更方便：

```
In [2]: from pandas import Series, DataFrame
```

5.1 pandas的数据结构介绍

要使用pandas，你首先就得熟悉它的两个主要数据结构：Series和DataFrame。虽然它们并不能解决所有问题，但它们为大多数应用提供了一种可靠的、易于使用的基础。

Series

Series是一种类似于一维数组的对象，它由一组数据（各种

NumPy数据类型）以及一组与之相关的数据标签（即索引）组成。仅由一组数据即可产生最简单的Series：

```
In [11]: obj = pd.Series([4, 7, -5, 3])

In [12]: obj
Out[12]:
0    4
1    7
2   -5
3    3
dtype: int64
```

Series的字符串表现形式为：索引在左边，值在右边。由于我们没有为数据指定索引，于是会自动创建一个0到N-1（N为数据的长度）的整数型索引。你可以通过Series 的values和index属性获取其数组表示形式和索引对象：

```
In [13]: obj.values
Out[13]: array([ 4,  7, -5,  3])

In [14]: obj.index  # like range(4)
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

通常，我们希望所创建的Series带有一个可以对各个数据点进行标记的索引：

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d',

In [16]: obj2
Out[16]:
d    4
b    7
a   -5
c    3
dtype: int64
```

```
In [17]: obj2.index
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

与普通NumPy数组相比，你可以通过索引的方式选取Series中的单个或一组值：

```
In [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c    3
a   -5
d    6
dtype: int64
```

['c', 'a', 'd']是索引列表，即使它包含的是字符串而不是整数。

使用NumPy函数或类似NumPy的运算（如根据布尔型数组进行过滤、标量乘法、应用数学函数等）都会保留索引值的链接：

```
In [21]: obj2[obj2 > 0]
Out[21]:
d    6
b    7
c    3
dtype: int64

In [22]: obj2 * 2
Out[22]:
d    12
b    14
a   -10
```

```
c     6
dtype: int64

In [23]: np.exp(obj2)
Out[23]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

还可以将Series看成是一个定长的有序字典，因为它是索引值到数据值的一个映射。它可以用在许多原本需要字典参数的函数中：

```
In [24]: 'b' in obj2
Out[24]: True

In [25]: 'e' in obj2
Out[25]: False
```

如果数据被存放在一个Python字典中，也可以直接通过这个字典来创建Series：

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Ore

In [27]: obj3 = pd.Series(sdata)

In [28]: obj3
Out[28]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64
```

如果只传入一个字典，则结果Series中的索引就是原字典的键（有序排列）。你可以传入排好序的字典的键以改变顺序：

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Te

In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

在这个例子中，sdata中跟states索引相匹配的那3个值会被找出来并放到相应的位置上，但由于"California"所对应的sdata值找不到，所以其结果就为NaN（即“非数字”（not a number），在pandas中，它用于表示缺失或NA值）。因为‘Utah’不在states中，它被从结果中除去。

我将使用缺失（missing）或NA表示缺失数据。pandas的isnull和notnull函数可用于检测缺失数据：

```
In [32]: pd.isnull(obj4)
Out[32]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool

In [33]: pd.notnull(obj4)
Out[33]:
California    False
Ohio           True
```

```
Oregon          True
Texas           True
dtype: bool
```

Series也有类似的实例方法：

```
In [34]: obj4.isnull()
Out[34]:
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

我将在第7章详细讲解如何处理缺失数据。

对于许多应用而言，Series最重要的一个功能是，它会根据运算的索引标签自动对齐数据：

```
In [35]: obj3
Out[35]:
Ohio      35000
Oregon    16000
Texas     71000
Utah       5000
dtype: int64

In [36]: obj4
Out[36]:
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64

In [37]: obj3 + obj4
Out[37]:
```

```
California          NaN
Ohio            70000.0
Oregon          32000.0
Texas          142000.0
Utah                NaN
dtype: float64
```

数据对齐功能将在后面详细讲解。如果你使用过数据库，你可以认为是类似join的操作。

Series对象本身及其索引都有一个name属性，该属性跟pandas其他的关键功能关系非常密切：

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'

In [40]: obj4
Out[40]:
state
California         NaN
Ohio           35000.0
Oregon         16000.0
Texas          71000.0
Name: population, dtype: float64
```

Series的索引可以通过赋值的方式就地修改：

```
In [41]: obj
Out[41]:
0    4
1    7
2   -5
3    3
dtype: int64

In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

```
In [43]: obj
Out[43]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

DataFrame

DataFrame是一个表格型的数据结构，它含有一组有序的列，每列可以是不同的值类型（数值、字符串、布尔值等）。DataFrame既有行索引也有列索引，它可以被看做由Series组成的字典（共用同一个索引）。DataFrame中的数据是以一个或多个二维块存放的（而不是列表、字典或别的一维数据结构）。有关DataFrame内部的技术细节远远超出了本书所讨论的范围。

> 笔记：虽然DataFrame是以二维结构保存数据的，但你仍然可以轻松地将其表示为更高维度的数据（层次化索引的表格型结构，这是pandas中许多高级数据处理功能的关键要素，我们会在第8章讨论这个问题）。

建DataFrame的办法有很多，最常用的一种是直接传入一个由等长列表或NumPy数组组成的字典：

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', '
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

结果DataFrame会自动加上索引（跟Series一样），且全部列会被有序排列：

```
In [45]: frame
```

```
Out[45]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
5  3.2  Nevada  2003
```

如果你使用的是Jupyter notebook，pandas DataFrame对象会以对浏览器友好的HTML表格的方式呈现。

对于特别大的DataFrame，head方法会选取前五行：

```
In [46]: frame.head()
Out[46]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

如果指定了列序列，则DataFrame的列就会按照指定顺序进行排列：

```
In [47]: pd.DataFrame(data, columns=['year', 'state',
Out[47]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

如果传入的列在数据中找不到，就会在结果中产生缺失值：

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 
   ....:                      index=['one', 'two', '
   ....:                             'five', 'six'])

In [49]: frame2
Out[49]: 
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN
six    2003  Nevada  3.2  NaN

In [50]: frame2.columns
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtyp
```

通过类似字典标记的方式或属性的方式，可以将DataFrame的列获取为一个Series：

```
In [51]: frame2['state']
Out[51]: 
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object

In [52]: frame2.year
Out[52]: 
one      2000
two      2001
```

```
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

> 笔记：IPython提供了类似属性的访问（即frame2.year）和tab补全。
> frame2[column]适用于任何列的名，但是frame2.column只有在列名是一个合理的Python变量名时才适用。

注意，返回的Series拥有原DataFrame相同的索引，且其name属性也已经被相应地设置好了。

行也可以通过位置或名称的方式进行获取，比如用loc属性（稍后将对此进行详细讲解）：

```
In [53]: frame2.loc['three']
Out[53]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name: three, dtype: object
```

列可以通过赋值的方式进行修改。例如，我们可以给那个空的"debt"列赋上一个标量值或一组值：

```
In [54]: frame2['debt'] = 16.5

In [55]: frame2
Out[55]:
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
```

```
five    2002  Nevada  2.9  16.5
six     2003  Nevada  3.2  16.5

In [56]: frame2['debt'] = np.arange(6.)

In [57]: frame2
Out[57]:
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   1.0
three  2002    Ohio  3.6   2.0
four   2001  Nevada  2.4   3.0
five   2002  Nevada  2.9   4.0
six    2003  Nevada  3.2   5.0
```

将列表或数组赋值给某个列时，其长度必须跟DataFrame的长度相匹配。如果赋值的是一个Series，就会精确匹配DataFrame的索引，所有的空位都将被填上缺失值：

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['

In [59]: frame2['debt'] = val

In [60]: frame2
Out[60]:
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
six    2003  Nevada  3.2   NaN
```

为不存在的列赋值会创建出一个新列。关键字del用于删除列。

作为del的例子，我先添加一个新的布尔值的列，state是否

为'Ohio'：

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'

In [62]: frame2
Out[62]:
       year   state  pop   debt  eastern
one    2000    Ohio  1.5    NaN     True
two    2001    Ohio  1.7   -1.2     True
three  2002    Ohio  3.6    NaN     True
four   2001  Nevada  2.4   -1.5    False
five   2002  Nevada  2.9   -1.7    False
six    2003  Nevada  3.2    NaN    False
```

> 注意：不能用frame2.eastern创建新的列。

del方法可以用来删除这列：

```
In [63]: del frame2['eastern']

In [64]: frame2.columns
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtyp
```

> 注意：通过索引方式返回的列只是相应数据的视图而已，并不是副本。因此，对返回的Series所做的任何就地修改全都会反映到源DataFrame上。通过Series的copy方法即可指定复制列。

另一种常见的数据形式是嵌套字典：

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
....:        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6
```

如果嵌套字典传给DataFrame，pandas就会被解释为：外层字典的键作为列，内层键则作为行索引：

```
In [66]: frame3 = pd.DataFrame(pop)

In [67]: frame3
Out[67]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
2002     2.9   3.6
```

你也可以使用类似NumPy数组的方法，对DataFrame进行转置（交换行和列）：

```
In [68]: frame3.T
Out[68]:
        2000  2001  2002
Nevada   NaN   2.4   2.9
Ohio     1.5   1.7   3.6
```

内层字典的键会被合并、排序以形成最终的索引。如果明确指定了索引，则不会这样：

```
In [69]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[69]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2003     NaN   NaN
```

由Series组成的字典差不多也是一样的用法：

```
In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
....:           'Nevada': frame3['Nevada'][:2]}

In [71]: pd.DataFrame(pdata)
Out[71]:
      Nevada  Ohio
2000     NaN   1.5
```

```
2001      2.4   1.7
```

表5-1列出了DataFrame构造函数所能接受的各种数据。

表5-1：可以输入给DataFrame构造器的数据

类型	说明
二维ndarray	数据矩阵，还可以传入行标和列标
由数组、列表或元组组成的字典	每个序列会变成DataFrame的一列。所有序列的长度必须相同
NumPy的结构化/记录数组	类似于"由数组组成的字典"
由Series组成的字典	每个Series会成为一列。如果没有显式指定索引，则各Series的索引会被合并成结果的行索引
由字典组成的字典	各内层字典会成为一列。键会被合并成结果的行索引，跟"由Series组成的字典"的情况一样
字典或Series的列表	各项将会成为DataFrame的一行。字典键或Series索引的并集将会成为DataFrame的列标
由列表或元组组成的列表	类似于"二维ndarray"
另一个DataFrame	该DataFrame的索引将会被沿用，除非显式指定了其他索引
NumPy的MaskedArray	类似于"二维ndarray"的情况，只是掩码值在结果DataFrame会变成NA/缺失值

如果设置了DataFrame的index和columns的name属性，则这些信息也会被显示出来：

```
In [72]: frame3.index.name = 'year'; frame3.columns.na

In [73]: frame3
Out[73]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
```

跟Series一样，values属性也会以二维ndarray的形式返回DataFrame中的数据：

```
In [74]: frame3.values
Out[74]:
array([[ nan,  1.5],
       [ 2.4,  1.7],
       [ 2.9,  3.6]])
```

如果DataFrame各列的数据类型不同，则值数组的dtype就会选用能兼容所有列的数据类型：

```
In [75]: frame2.values
Out[75]:
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2, nan]], dtype=object)
```

索引对象

pandas的索引对象负责管理轴标签和其他元数据（比如轴名称等）。构建Series或DataFrame时，所用到的任何数组或其他序列的标签都会被转换成一个Index：

```
In [76]: obj = pd.Series(range(3), index=['a', 'b', '

In [77]: index = obj.index

In [78]: index
Out[78]: Index(['a', 'b', 'c'], dtype='object')

In [79]: index[1:]
Out[79]: Index(['b', 'c'], dtype='object')
```

Index对象是不可变的，因此用户不能对其进行修改：

```
index[1] = 'd'  # TypeError
```

不可变可以使Index对象在多个数据结构之间安全共享：

```
In [80]: labels = pd.Index(np.arange(3))

In [81]: labels
Out[81]: Int64Index([0, 1, 2], dtype='int64')

In [82]: obj2 = pd.Series([1.5, -2.5, 0], index=label

In [83]: obj2
Out[83]:
0    1.5
1   -2.5
2    0.0
dtype: float64

In [84]: obj2.index is labels
Out[84]: True
```

> 注意：虽然用户不需要经常使用Index的功能，但是因为一些操作会生成包含被索引化的数据，理解它们的工作原理是很重要的。

除了类似于数组，Index的功能也类似一个固定大小的集合：

```
In [85]: frame3
Out[85]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
In [86]: frame3.columns
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', na
```

```
In [87]: 'Ohio' in frame3.columns
Out[87]: True

In [88]: 2003 in frame3.index
Out[88]: False
```

与python的集合不同，pandas的Index可以包含重复的标签：

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar',

In [90]: dup_labels
Out[90]: Index(['foo', 'foo', 'bar', 'bar'], dtype='o
```

选择重复的标签，会显示所有的结果。

每个索引都有一些方法和属性，它们可用于设置逻辑并回答有关该索引所包含的数据的常见问题。表5-2列出了这些函数。

表5-2：Index的方法和属性

方法	说明
append	连接另一个Index对象，产生一个新的Index
difference	计算差集，并得到一个Index
intersection	计算交集
union	计算并集
isin	计算一个指示各值是否都包含在参数集合中的布尔型数组
delete	删除索引i处的元素，并得到新的Index
drop	删除传入的值，并得到新的Index
insert	将元素插入到索引i处，并得到新的Index
is_monotonic	当各元素均大于等于前一个元素时，返回True
is_unique	当Index没有重复值时，返回True
unique	计算Index中唯一值的数组

5.2 基本功能

本节中，我将介绍操作Series和DataFrame中的数据的基本手

段。后续章节将更加深入地挖掘pandas在数据分析和处理方面的功能。本书不是pandas库的详尽文档，主要关注的是最重要的功能，那些不大常用的内容（也就是那些更深奥的内容）就交给你自己去摸索吧。

重新索引

pandas对象的一个重要方法是reindex，其作用是创建一个新对象，它的数据符合新的索引。看下面的例子：

```
In [91]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=

In [92]: obj
Out[92]:
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

用该Series的reindex将会根据新索引进行重排。如果某个索引值当前不存在，就引入缺失值：

```
In [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e']

In [94]: obj2
Out[94]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

对于时间序列这样的有序数据，重新索引时可能需要做一些插值

处理。method选项即可达到此目的，例如，使用ffill可以实现前向值填充：

```
In [95]: obj3 = pd.Series(['blue', 'purple', 'yellow'

In [96]: obj3
Out[96]:
0      blue
2    purple
4    yellow
dtype: object

In [97]: obj3.reindex(range(6), method='ffill')
Out[97]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

借助DataFrame，reindex可以修改（行）索引和列。只传递一个序列时，会重新索引结果的行：

```
In [98]: frame = pd.DataFrame(np.arange(9).reshape((3
   ....:                      index=['a', 'c', 'd'],
   ....:                      columns=['Ohio', 'Texas

In [99]: frame
Out[99]:
   Ohio  Texas  California
a     0      1           2
c     3      4           5
d     6      7           8
```

```
In [100]: frame2 = frame.reindex(['a', 'b', 'c', 'd']

In [101]: frame2
Out[101]:
   Ohio  Texas  California
a   0.0    1.0         2.0
b   NaN    NaN         NaN
c   3.0    4.0         5.0
d   6.0    7.0         8.0
```

列可以用columns关键字重新索引：

```
In [102]: states = ['Texas', 'Utah', 'California']

In [103]: frame.reindex(columns=states)
Out[103]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

表5-3列出了reindex函数的各参数及说明。

表5-3：reindex函数的参数

参数	说明
index	用作索引的新序列。既可以是Index实例，也可以是其他序列型的Python数据结构。Index会被完全使用，就像没有任何复制一样
method	插值（填充）方式，具体参数请参见表5-4
fill_value	在重新索引的过程中，需要引入缺失值时使用的替代值
limit	前向或后向填充时的最大填充量
tolerance	向前后向后填充时，填充不准确匹配项的最大间距（绝对值距离）
level	在MultiIndex的指定级别上匹配简单索引，否则选取其子集
copy	默认为True，无论如何都复制；如果为False，则新旧相等就不复制

丢弃指定轴上的项

丢弃某条轴上的一个或多个项很简单，只要有一个索引数组或列表即可。由于需要执行一些数据整理和集合逻辑，所以drop方法返回的是一个在指定轴上删除了指定值的新对象：

```
In [105]: obj = pd.Series(np.arange(5.), index=['a',

In [106]: obj
Out[106]:
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64

In [107]: new_obj = obj.drop('c')

In [108]: new_obj
Out[108]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64

In [109]: obj.drop(['d', 'c'])
Out[109]:
a    0.0
b    1.0
e    4.0
dtype: float64
```

对于DataFrame，可以删除任意轴上的索引值。为了演示，先新建一个DataFrame例子：

```
In [110]: data = pd.DataFrame(np.arange(16).reshape((
```

```
   .....:                      index=['Ohio', 'Colorad
   .....:                      columns=['one', 'two',

In [111]: data
Out[111]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

用标签序列调用drop会从行标签（axis 0）删除值：

```
In [112]: data.drop(['Colorado', 'Ohio'])
Out[112]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

通过传递axis=1或axis='columns'可以删除列的值：

```
In [113]: data.drop('two', axis=1)
Out[113]:
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [114]: data.drop(['two', 'four'], axis='columns')
Out[114]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

许多函数，如drop，会修改Series或DataFrame的大小或形状，可以就地修改对象，不会返回新的对象：

```
In [115]: obj.drop('c', inplace=True)

In [116]: obj
Out[116]:
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

小心使用inplace，它会销毁所有被删除的数据。

索引、选取和过滤

Series索引（obj[...]）的工作方式类似于NumPy数组的索引，只不过Series的索引值不只是整数。下面是几个例子：

```
In [117]: obj = pd.Series(np.arange(4.), index=['a',

In [118]: obj
Out[118]:
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

In [119]: obj['b']
Out[119]: 1.0

In [120]: obj[1]
Out[120]: 1.0

In [121]: obj[2:4]
```

```
Out[121]:
c    2.0
d    3.0
dtype: float64

In [122]: obj[['b', 'a', 'd']]
Out[122]:
b    1.0
a    0.0
d    3.0
dtype: float64

In [123]: obj[[1, 3]]
Out[123]:
b    1.0
d    3.0
dtype: float64

In [124]: obj[obj < 2]
Out[124]:
a    0.0
b    1.0
dtype: float64
```

利用标签的切片运算与普通的Python切片运算不同，其末端是包含的：

```
In [125]: obj['b':'c']
Out[125]:
b    1.0
c    2.0
dtype: float64
```

用切片可以对Series的相应部分进行设置：

```
In [126]: obj['b':'c'] = 5
```

```
In [127]: obj
Out[127]:
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

用一个值或序列对DataFrame进行索引其实就是获取一个或多个列:

```
In [128]: data = pd.DataFrame(np.arange(16).reshape((
   .....:                      index=['Ohio', 'Colorad
   .....:                      columns=['one', 'two',

In [129]: data
Out[129]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [130]: data['two']
Out[130]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64

In [131]: data[['three', 'one']]
Out[131]:
          three  one
Ohio          2    0
Colorado      6    4
```

```
Utah          10    8
New York      14   12
```

这种索引方式有几个特殊的情况。首先通过切片或布尔型数组选取数据：

```
In [132]: data[:2]
Out[132]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [133]: data[data['three'] > 5]
Out[133]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

选取行的语法data[:2]十分方便。向[ ]传递单一的元素或列表，就可选择列。

另一种用法是通过布尔型DataFrame（比如下面这个由标量比较运算得出的）进行索引：

```
In [134]: data < 5
Out[134]:
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False

In [135]: data[data < 5] = 0

In [136]: data
```

```
Out[136]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

这使得DataFrame的语法与NumPy二维数组的语法很像。

用loc和iloc进行选取

对于DataFrame的行的标签索引，我引入了特殊的标签运算符loc和iloc。它们可以让你用类似NumPy的标记，使用轴标签（loc）或整数索引（iloc），从DataFrame选择行和列的子集。

作为一个初步示例，让我们通过标签选择一行和多列：

```
In [137]: data.loc['Colorado', ['two', 'three']]
Out[137]:
two      5
three    6
Name: Colorado, dtype: int64
```

然后用iloc和整数进行选取：

```
In [138]: data.iloc[2, [3, 0, 1]]
Out[138]:
four    11
one      8
two      9
Name: Utah, dtype: int64

In [139]: data.iloc[2]
Out[139]:
one       8
two       9
three    10
```

```
four     11
Name: Utah, dtype: int64

In [140]: data.iloc[[1, 2], [3, 0, 1]]
Out[140]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9
```

这两个索引函数也适用于一个标签或多个标签的切片：

```
In [141]: data.loc[:'Utah', 'two']
Out[141]:
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int64

In [142]: data.iloc[:, :3][data.three > 5]
Out[142]:
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
```

所以，在pandas中，有多个方法可以选取和重新组合数据。对于DataFrame，表5-4进行了总结。后面会看到，还有更多的方法进行层级化索引。

> 笔记：在一开始设计pandas时，我觉得用frame[:, col]选取列过于繁琐（也容易出错），因为列的选择是非常常见的操作。我做了些取舍，将花式索引的功能（标签和整数）放到了ix运算符中。在实践中，这会导致许多边缘情况，数据的轴标签是整数，所以pandas团队决定创造loc和iloc运算符分别处理严格基于标签和整数的索引。
> ix运算符仍然可用，但并不推荐。

类型	说明
df[val]	从 DataFrame 选取单列或一组列；在特殊情况下比较便利：布尔型数组（过滤行）、切片（行切片）、或布尔型 DataFrame（根据条件设置值）
df.loc[val]	通过标签，选取 DataFrame 的单个行或一组行
df.loc[:, val]	通过标签，选取单列或列子集
df.loc[val1, val2]	通过标签，同时选取行和列
df.iloc[where]	通过整数位置，从 DataFrame 选取单个行或行子集
df.iloc[:, where]	通过整数位置，从 DataFrame 选取单个列或列子集
df.iloc[where_i, where_j]	通过整数位置，同时选取行和列
df.at[label_i, label_j]	通过行和列标签，选取单一的标量
df.iat[i, j]	通过行和列的位置（整数），选取单一的标量
reindex	通过标签选取行或列
get_value, set_value	通过行和列标签选取单一值

表5-4 DataFrame的索引选项

整数索引

处理整数索引的pandas对象常常难住新手，因为它与Python内置的列表和元组的索引语法不同。例如，你可能不认为下面的代码会出错：

```
ser = pd.Series(np.arange(3.))
ser
ser[-1]
```

这里，pandas可以勉强进行整数索引，但是会导致小bug。我们有包含0,1,2的索引，但是引入用户想要的东西（基于标签或位置的索引）很难：

```
In [144]: ser
Out[144]:
0    0.0
1    1.0
2    2.0
dtype: float64
```

另外，对于非整数索引，不会产生歧义：

```
In [145]: ser2 = pd.Series(np.arange(3.), index=['a',

In [146]: ser2[-1]
Out[146]: 2.0
```

为了进行统一，如果轴索引含有整数，数据选取总会使用标签。为了更准确，请使用loc（标签）或iloc（整数）：

```
In [147]: ser[:1]
Out[147]:
0    0.0
dtype: float64

In [148]: ser.loc[:1]
Out[148]:
0    0.0
1    1.0
dtype: float64

In [149]: ser.iloc[:1]
Out[149]:
0    0.0
dtype: float64
```

算术运算和数据对齐

pandas最重要的一个功能是，它可以对不同索引的对象进行算术运算。在将对象相加时，如果存在不同的索引对，则结果的索引就是该索引对的并集。对于有数据库经验的用户，这就像在索引标签上进行自动外连接。看一个简单的例子：

```
In [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=

In [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
```

```
   .....:                index=['a', 'c', 'e', 'f', '

In [152]: s1
Out[152]:
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64

In [153]: s2
Out[153]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

将它们相加就会产生：

```
In [154]: s1 + s2
Out[154]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

自动的数据对齐操作在不重叠的索引处引入了NA值。缺失值会在算术运算过程中传播。

对于DataFrame，对齐操作会同时发生在行和列上：

```
In [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3
```

```
   .....:                          index=['Ohio', 'Texas',

In [156]: df2 = pd.DataFrame(np.arange(12.).reshape((
   .....:                          index=['Utah', 'Ohio', '

In [157]: df1
Out[157]:
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0

In [158]: df2
Out[158]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

把它们相加后将会返回一个新的DataFrame，其索引和列为原来那两个DataFrame的并集：

```
In [159]: df1 + df2
Out[159]:
            b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

因为'c'和'e'列均不在两个DataFrame对象中，在结果中以缺省值呈现。行也是同样。

如果DataFrame对象相加，没有共用的列或行标签，结果都会是

空:

```
In [160]: df1 = pd.DataFrame({'A': [1, 2]})

In [161]: df2 = pd.DataFrame({'B': [3, 4]})

In [162]: df1
Out[162]:
   A
0  1
1  2

In [163]: df2
Out[163]:
   B
0  3
1  4

In [164]: df1 - df2
Out[164]:
    A   B
0 NaN NaN
1 NaN NaN
```

在算术方法中填充值

在对不同索引的对象进行算术运算时，你可能希望当一个对象中某个轴标签在另一个对象中找不到时填充一个特殊值（比如0）：

```
In [165]: df1 = pd.DataFrame(np.arange(12.).reshape((
   .....:                          columns=list('abcd'))

In [166]: df2 = pd.DataFrame(np.arange(20.).reshape((
   .....:                          columns=list('abcde'))

In [167]: df2.loc[1, 'b'] = np.nan
```

```
In [168]: df1
Out[168]:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0

In [169]: df2
Out[169]:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   NaN   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

将它们相加时，没有重叠的位置就会产生NA值：

```
In [170]: df1 + df2
Out[170]:
      a     b     c     d   e
0   0.0   2.0   4.0   6.0 NaN
1   9.0   NaN  13.0  15.0 NaN
2  18.0  20.0  22.0  24.0 NaN
3   NaN   NaN   NaN   NaN NaN

使用df1的add方法，传入df2以及一个fill_value参数：
```python
In [171]: df1.add(df2, fill_value=0)
Out[171]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```
```

表5-5列出了Series和DataFrame的算术方法。它们每个都有一个副本，以字母r开头，它会翻转参数。因此这两个语句是等价的：

```
In [172]: 1 / df1
Out[172]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909

In [173]: df1.rdiv(1)
Out[173]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909
```

方法	说明
add, radd	用于加法（+）的方法
sub, rsub	用于减法（-）的方法
div, rdiv	用于除法（/）的方法
floordiv, rfloordiv	用于底除（//）的方法
mul, rmul	用于乘法（*）的方法
pow, rpow	用于指数（**）的方法

表5-5 灵活的算术方法

与此类似，在对Series或DataFrame重新索引时，也可以指定一个填充值：

```
In [174]: df1.reindex(columns=df2.columns, fill_value
Out[174]:
     a    b     c     d   e
```

```
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

DataFrame和Series之间的运算

跟不同维度的NumPy数组一样，DataFrame和Series之间算术运算也是有明确规定的。先来看一个具有启发性的例子，计算一个二维数组与其某行之间的差：

```
In [175]: arr = np.arange(12.).reshape((3, 4))

In [176]: arr
Out[176]:
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])

In [177]: arr[0]
Out[177]: array([ 0.,  1.,  2.,  3.])

In [178]: arr - arr[0]
Out[178]:
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

当我们从arr减去arr[0]，每一行都会执行这个操作。这就叫做广播（broadcasting），附录A将对此进行详细讲解。DataFrame和Series之间的运算差不多也是如此：

```
In [179]: frame = pd.DataFrame(np.arange(12.).reshape
   .....:                      columns=list('bde'),
   .....:                      index=['Utah', 'Ohio',

In [180]: series = frame.iloc[0]
```

```
In [181]: frame
Out[181]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [182]: series
Out[182]:
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

默认情况下，DataFrame和Series之间的算术运算会将Series的索引匹配到DataFrame的列，然后沿着行一直向下广播：

```
In [183]: frame - series
Out[183]:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

如果某个索引值在DataFrame的列或Series的索引中找不到，则参与运算的两个对象就会被重新索引以形成并集：

```
In [184]: series2 = pd.Series(range(3), index=['b', '

In [185]: frame + series2
Out[185]:
          b   d     e   f
Utah    0.0 NaN   3.0 NaN
```

```
Ohio     3.0 NaN   6.0 NaN
Texas    6.0 NaN   9.0 NaN
Oregon   9.0 NaN  12.0 NaN
```

如果你希望匹配行且在列上广播，则必须使用算术运算方法。例如：

```
In [186]: series3 = frame['d']

In [187]: frame
Out[187]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [188]: series3
Out[188]:
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64

In [189]: frame.sub(series3, axis='index')
Out[189]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

传入的轴号就是希望匹配的轴。在本例中，我们的目的是匹配DataFrame的行索引（axis='index' or axis=0）并进行广播。

函数应用和映射

NumPy的ufuncs（元素级数组方法）也可用于操作pandas对象：

```
In [190]: frame = pd.DataFrame(np.random.randn(4, 3),
   .....:                      index=['Utah', 'Ohio',

In [191]: frame
Out[191]:
               b         d         e
Utah   -0.204708  0.478943 -0.519439
Ohio   -0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221

In [192]: np.abs(frame)
Out[192]:
               b         d         e
Utah    0.204708  0.478943  0.519439
Ohio    0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221
```

另一个常见的操作是，将函数应用到由各列或行所形成的一维数组上。DataFrame的apply方法即可实现此功能：

```
In [193]: f = lambda x: x.max() - x.min()

In [194]: frame.apply(f)
Out[194]:
b    1.802165
d    1.684034
e    2.689627
dtype: float64
```

这里的函数f，计算了一个Series的最大和虽小的差，在frane的每列都执行了一次。结果是一个Series，使用frame的列作为索引。

如果传递axis='columns'到apply，这个函数会在每行执行：

```
In [195]: frame.apply(f, axis='columns')
Out[195]:
Utah      0.998382
Ohio      2.521511
Texas     0.676115
Oregon    2.542656
dtype: float64
```

许多最为常见的数组统计功能都被实现成DataFrame的方法（如sum和mean），因此无需使用apply方法。

传递到apply的函数不是必须返回一个标量，还可以返回由多个值组成的Series：

```
In [196]: def f(x):
   .....:     return pd.Series([x.min(), x.max()], in

In [197]: frame.apply(f)
Out[197]:
            b         d         e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

元素级的Python函数也是可以用的。假如你想得到frame中各个浮点值的格式化字符串，使用applymap即可：

```
In [198]: format = lambda x: '%.2f' % x

In [199]: frame.applymap(format)
Out[199]:
```

```
              b     d      e
Utah    -0.20  0.48  -0.52
Ohio    -0.56  1.97   1.39
Texas    0.09  0.28   0.77
Oregon   1.25  1.01  -1.30
```

之所以叫做applymap，是因为Series有一个用于应用元素级函数的map方法：

```
In [200]: frame['e'].map(format)
Out[200]:
Utah      -0.52
Ohio       1.39
Texas      0.77
Oregon    -1.30
Name: e, dtype: object
```

排序和排名

根据条件对数据集排序（sorting）也是一种重要的内置运算。要对行或列索引进行排序（按字典顺序），可使用sort_index方法，它将返回一个已排序的新对象：

```
In [201]: obj = pd.Series(range(4), index=['d', 'a',

In [202]: obj.sort_index()
Out[202]:
a    1
b    2
c    3
d    0
dtype: int64
```

对于DataFrame，则可以根据任意一个轴上的索引进行排序：

```
In [203]: frame = pd.DataFrame(np.arange(8).reshape((
```

```
   .....:                          index=['three', 'one']
   .....:                          columns=['d', 'a', 'b'

In [204]: frame.sort_index()
Out[204]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [205]: frame.sort_index(axis=1)
Out[205]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

数据默认是按升序排序的，但也可以降序排序：

```
In [206]: frame.sort_index(axis=1, ascending=False)
Out[206]:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

若要按值对Series进行排序，可使用其sort_values方法：

```
In [207]: obj = pd.Series([4, 7, -3, 2])

In [208]: obj.sort_values()
Out[208]:
2   -3
3    2
0    4
1    7
dtype: int64
```

在排序时，任何缺失值默认都会被放到Series的末尾：

```
In [209]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 

In [210]: obj.sort_values()
Out[210]:
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

当排序一个DataFrame时，你可能希望根据一个或多个列中的值进行排序。将一个或多个列的名字传递给sort_values的by选项即可达到该目的：

```
In [211]: frame = pd.DataFrame({'b': [4, 7, -3, 2], '

In [212]: frame
Out[212]:
   a  b
0  0  4
1  1  7
2  0 -3
3  1  2

In [213]: frame.sort_values(by='b')
Out[213]:
   a  b
2  0 -3
3  1  2
0  0  4
1  1  7
```

要根据多个列进行排序，传入名称的列表即可：

```
In [214]: frame.sort_values(by=['a', 'b'])
Out[214]:
   a  b
2  0 -3
0  0  4
3  1  2
1  1  7
```

排名会从1开始一直到数组中有效数据的数量。接下来介绍Series和DataFrame的rank方法。默认情况下，rank是通过“为各组分配一个平均排名”的方式破坏平级关系的：

```
In [215]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
In [216]: obj.rank()
Out[216]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

也可以根据值在原数据中出现的顺序给出排名：

```
In [217]: obj.rank(method='first')
Out[217]:
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

这里，条目0和2没有使用平均排名6.5，它们被设成了6和7，因为数据中标签0位于标签2的前面。

你也可以按降序进行排名：

```
# Assign tie values the maximum rank in the group
In [218]: obj.rank(ascending=False, method='max')
Out[218]:
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

表5-6列出了所有用于破坏平级关系的method选项。DataFrame可以在行或列上计算排名：

```
In [219]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2],
   .....:                       'c': [-2, 5, 8, -2.5]

In [220]: frame
Out[220]:
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5

In [221]: frame.rank(axis='columns')
Out[221]:
     a    b    c
0  2.0  3.0  1.0
1  1.0  3.0  2.0
2  2.0  1.0  3.0
```

```
3  2.0  3.0  1.0
```

方法	说明
'average'	默认：在相等分组中，为各个值分配平均排名
'min'	使用整个分组的最小排名
'max'	使用整个分组的最大排名
'first'	按值在原始数据中的出现顺序分配排名
'dense'	类似于'min'方法，但是排名总是在组间增加1，而不是组中相同的元素数

表5-6 排名时用于破坏平级关系的方法

带有重复标签的轴索引

直到目前为止，我所介绍的所有范例都有着唯一的轴标签（索引值）。虽然许多pandas函数（如reindex）都要求标签唯一，但这并不是强制性的。我们来看看下面这个简单的带有重复索引值的Series：

```
In [222]: obj = pd.Series(range(5), index=['a', 'a',

In [223]: obj
Out[223]:
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

索引的is_unique属性可以告诉你它的值是否是唯一的：

```
In [224]: obj.index.is_unique
Out[224]: False
```

对于带有重复值的索引，数据选取的行为将会有些不同。如果某

个索引对应多个值，则返回一个Series；而对应单个值的，则返回一个标量值：

```
In [225]: obj['a']
Out[225]:
a    0
a    1
dtype: int64

In [226]: obj['c']
Out[226]: 4
```

这样会使代码变复杂，因为索引的输出类型会根据标签是否有重复发生变化。

对DataFrame的行进行索引时也是如此：

```
In [227]: df = pd.DataFrame(np.random.randn(4, 3), in

In [228]: df
Out[228]:
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [229]: df.loc['b']
Out[229]:
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

5.3 汇总和计算描述统计

pandas对象拥有一组常用的数学和统计方法。它们大部分都属

于约简和汇总统计，用于从Series中提取单个值（如sum或mean）或从DataFrame的行或列中提取一个Series。跟对应的NumPy数组方法相比，它们都是基于没有缺失数据的假设而构建的。看一个简单的DataFrame：

```
In [230]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.
   .....:                     [np.nan, np.nan], [0.75,
   .....:                    index=['a', 'b', 'c', 'd'
   .....:                    columns=['one', 'two'])

In [231]: df
Out[231]:
    one  two
a  1.40  NaN
b  7.10 -4.5
c   NaN  NaN
d  0.75 -1.3
```

调用DataFrame的sum方法将会返回一个含有列的和的Series：

```
In [232]: df.sum()
Out[232]:
one    9.25
two   -5.80
dtype: float64
```

传入axis='columns'或axis=1将会按行进行求和运算：

```
In [233]: df.sum(axis=1)
Out[233]:
a    1.40
b    2.60
c     NaN
d   -0.55
```

NA值会自动被排除，除非整个切片（这里指的是行或列）都是

NA。通过skipna选项可以禁用该功能：

```
In [234]: df.mean(axis='columns', skipna=False)
Out[234]:
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

表5-7列出了这些约简方法的常用选项。

表5-7：约简方法的选项

选项	说明
axis	约简的轴。DataFrame的行用0，列用1
skipna	排除缺失值，默认值为True
level	如果轴是层次化索引的（即MultiIndex），则根据level分组约简

有些方法（如idxmin和idxmax）返回的是间接统计（比如达到最小值或最大值的索引）：

```
In [235]: df.idxmax()
Out[235]:
one    b
two    d
dtype: object
```

另一些方法则是累计型的：

```
In [236]: df.cumsum()
Out[236]:
    one  two
a  1.40  NaN
b  8.50 -4.5
c   NaN  NaN
d  9.25 -5.8
```

还有一种方法，它既不是约简型也不是累计型。describe就是一个例子，它用于一次性产生多个汇总统计：

```
In [237]: df.describe()
Out[237]:
            one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

对于非数值型数据，describe会产生另外一种汇总统计：

```
In [238]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

In [239]: obj.describe()
Out[239]:
count     16
unique     3
top        a
freq       8
dtype: object
```

表5-8列出了所有与描述统计相关的方法。

表5-8 ：描述和汇总统计

方法	说明
count	非NA值的数量
describe	针对Series或各DataFrame列计算汇总统计
min、max	计算最小值和最大值
argmin、argmax	计算能够获取到最小值和最大值的索引位置（整数）
idxmin、idxmax	计算能够获取到最小值和最大值的索引值
quantile	计算样本的分位数（0到1）
sum	值的总和
mean	值的平均数
median	值的算术中位数（50%分位数）
mad	根据平均值计算平均绝对离差
var	样本值的方差
std	样本值的标准差
skew	样本值的偏度（三阶矩）
kurt	样本值的峰度（四阶矩）
cumsum	样本值的累计和
cummin、cummax	样本值的累计最大值和累计最小值
cumprod	样本值的累计积
diff	计算一阶差分（对时间序列很有用）
pct_change	计算百分数变化

相关系数与协方差

有些汇总统计（如相关系数和协方差）是通过参数对计算出来的。我们来看几个DataFrame，它们的数据来自Yahoo!Finance的股票价格和成交量，使用的是pandas-datareader包（可以用conda或pip安装）：

```
conda install pandas-datareader
```

我使用pandas_datareader模块下载了一些股票数据：

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GO
```

```
price = pd.DataFrame({ticker: data['Adj Close']
                      for ticker, data in all_data.ite
volume = pd.DataFrame({ticker: data['Volume']
                       for ticker, data in all_data.it
```

> 注意：此时Yahoo! Finance已经不存在了，因为2017年Yahoo!被Verizon收购了。参阅pandas-datareader文档，可以学习最新的功能。

现在计算价格的百分数变化，时间序列的操作会在第11章介绍：

```
In [242]: returns = price.pct_change()

In [243]: returns.tail()
Out[243]:
                AAPL      GOOG       IBM      MSFT
Date
2016-10-17 -0.000680  0.001837  0.002072 -0.003483
2016-10-18 -0.000681  0.019616 -0.026168  0.007690
2016-10-19 -0.002979  0.007846  0.003583 -0.002255
2016-10-20 -0.000512 -0.005652  0.001719 -0.004867
2016-10-21 -0.003930  0.003011 -0.012474  0.042096
```

Series的corr方法用于计算两个Series中重叠的、非NA的、按索引对齐的值的相关系数。与此类似，cov用于计算协方差：

```
In [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144

In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05
```

因为MSTF是一个合理的Python属性，我们还可以用更简洁的语法选择列：

```
In [246]: returns.MSFT.corr(returns.IBM)
Out[246]: 0.49976361144151144
```

另一方面，DataFrame的corr和cov方法将以DataFrame的形式分别返回完整的相关系数或协方差矩阵：

```
In [247]: returns.corr()
Out[247]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.407919  0.386817  0.389695
GOOG  0.407919  1.000000  0.405099  0.465919
IBM   0.386817  0.405099  1.000000  0.499764
MSFT  0.389695  0.465919  0.499764  1.000000

In [248]: returns.cov()
Out[248]:
          AAPL      GOOG       IBM      MSFT
AAPL  0.000277  0.000107  0.000078  0.000095
GOOG  0.000107  0.000251  0.000078  0.000108
IBM   0.000078  0.000078  0.000146  0.000089
MSFT  0.000095  0.000108  0.000089  0.000215
```

利用DataFrame的corrwith方法，你可以计算其列或行跟另一个Series或DataFrame之间的相关系数。传入一个Series将会返回一个相关系数值Series（针对各列进行计算）：

```
In [249]: returns.corrwith(returns.IBM)
Out[249]:
AAPL    0.386817
GOOG    0.405099
IBM     1.000000
MSFT    0.499764
dtype: float64
```

传入一个DataFrame则会计算按列名配对的相关系数。这里，我计算百分比变化与成交量的相关系数：

```
In [250]: returns.corrwith(volume)
Out[250]:
AAPL   -0.075565
GOOG   -0.007067
IBM    -0.204849
MSFT   -0.092950
dtype: float64
```

传入axis='columns'即可按行进行计算。无论如何，在计算相关系数之前，所有的数据项都会按标签对齐。

唯一值、值计数以及成员资格

还有一类方法可以从一维Series的值中抽取信息。看下面的例子：

```
In [251]: obj = pd.Series(['c', 'a', 'd', 'a', 'a', '
```

第一个函数是unique，它可以得到Series中的唯一值数组：

```
In [252]: uniques = obj.unique()

In [253]: uniques
Out[253]: array(['c', 'a', 'd', 'b'], dtype=object)
```

返回的唯一值是未排序的，如果需要的话，可以对结果再次进行排序（uniques.sort()）。相似的，value_counts用于计算一个Series中各值出现的频率：

```
In [254]: obj.value_counts()
Out[254]:
c    3
a    3
b    2
d    1
dtype: int64
```

为了便于查看，结果Series是按值频率降序排列的。value_counts还是一个顶级pandas方法，可用于任何数组或序列：

```
In [255]: pd.value_counts(obj.values, sort=False)
Out[255]:
a    3
b    2
c    3
d    1
dtype: int64
```

isin用于判断矢量化集合的成员资格，可用于过滤Series中或DataFrame列中数据的子集：

```
In [256]: obj
Out[256]:
0    c
1    a
2    d
3    a
4    a
5    b
6    b
7    c
8    c
dtype: object

In [257]: mask = obj.isin(['b', 'c'])

In [258]: mask
Out[258]:
0     True
1    False
2    False
3    False
4    False
```

```
5     True
6     True
7     True
8     True
dtype: bool

In [259]: obj[mask]
Out[259]:
0    c
5    b
6    b
7    c
8    c
dtype: object
```

与isin类似的是Index.get_indexer方法，它可以给你一个索引数组，从可能包含重复值的数组到另一个不同值的数组：

```
In [260]: to_match = pd.Series(['c', 'a', 'b', 'b', '

In [261]: unique_vals = pd.Series(['c', 'b', 'a'])

In [262]: pd.Index(unique_vals).get_indexer(to_match)
Out[262]: array([0, 2, 1, 1, 0, 2])
```

表5-9给出了这几个方法的一些参考信息。

方法	说明
isin	计算一个表示“Series 各值是否包含于传入的值序列中”的布尔型数组
match	计算一个数组中的各值到另一个不同值数组的整数索引；对于数据对齐和连接类型的操作十分有用
unique	计算 Series 中的唯一值数组，按发现的顺序返回
value_counts	返回一个 Series，其索引为唯一值，其值为频率，按计数值降序排列

表5-9 唯一值、值计数、成员资格方法

有时，你可能希望得到DataFrame中多个相关列的一张柱状图。例如：

```
In [263]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4]
   .....:                      'Qu2': [2, 3, 1, 2, 3]
   .....:                      'Qu3': [1, 5, 2, 4, 4]

In [264]: data
Out[264]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

将pandas.value_counts传给该DataFrame的apply函数，就会出现：

```
In [265]: result = data.apply(pd.value_counts).fillna

In [266]: result
Out[266]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

这里，结果中的行标签是所有列的唯一值。这些值是每个列中这些值的相应计数。

5.4 总结

在下一章，我们将讨论用pandas读取（或加载）和写入数据集的工具。

之后，我们将更深入地研究使用pandas进行数据清洗、规整、

分析和可视化工具。

第6章 数据加载、存储与文件格式

访问数据是使用本书所介绍的这些工具的第一步。我会着重介绍pandas的数据输入与输出，虽然别的库中也有不少以此为目的的工具。

输入输出通常可以划分为几个大类：读取文本文件和其他更高效的磁盘存储格式，加载数据库中的数据，利用Web API操作网络资源。

6.1 读写文本格式的数据

pandas提供了一些用于将表格型数据读取为DataFrame对象的函数。表6-1对它们进行了总结，其中read_csv和read_table可能会是你今后用得最多的。

函数	说明
read_csv	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为逗号
read_table	从文件、URL、文件型对象中加载带分隔符的数据。默认分隔符为制表符('\t')
read_fwf	读取定宽列格式数据（也就是说，没有分隔符）
read_clipboard	读取剪贴板中的数据，可以看做 read_table 的剪贴板版。再将网页转换为表格时很有用
read_excel	从 Excel XLS 或 XLSX file 读取表格数据
read_hdf	读取 pandas 写的 HDF5 文件
read_html	读取 HTML 文档中的所有表格
read_json	读取 JSON (JavaScript Object Notation)字符串中的数据
read_msgpack	二进制格式编码的 pandas 数据
read_pickle	读取 Python pickle 格式中存储的任意对象
read_sas	读取存储于 SAS 系统自定义存储格式的 SAS 数据集
read_sql	（使用 SQLAlchemy）读取 SQL 查询结果为 pandas 的 DataFrame
read_stata	读取 Stata 文件格式的数据集
read_feather	读取 Feather 二进制文件格式

表6-1 pandas中的解析函数

我将大致介绍一下这些函数在将文本数据转换为DataFrame时所用到的一些技术。这些函数的选项可以划分为以下几个大类：

- 索引：将一个或多个列当做返回的DataFrame处理，以及是否从文件、用户获取列名。
- 类型推断和数据转换：包括用户定义值的转换、和自定义的缺失值标记列表等。
- 日期解析：包括组合功能，比如将分散在多个列中的日期时间信息组合成结果中的单个列。
- 迭代：支持对大文件进行逐块迭代。
- 不规整数据问题：跳过一些行、页脚、注释或其他一些不重要的东西（比如由成千上万个逗号隔开的数值数据）。

因为工作中实际碰到的数据可能十分混乱，一些数据加载函数（尤其是read_csv）的选项逐渐变得复杂起来。面对不同的参数，感到头痛很正常（read_csv有超过50个参数）。pandas文档有这些参数的例子，如果你感到阅读某个文件很难，可以通过相似的足够多的例子找到正确的参数。

其中一些函数，比如pandas.read_csv，有类型推断功能，因为列数据的类型不属于数据类型。也就是说，你不需要指定列的类型到底是数值、整数、布尔值，还是字符串。其它的数据格式，如HDF5、Feather和msgpack，会在格式中存储数据类型。

日期和其他自定义类型的处理需要多花点工夫才行。首先我们来看一个以逗号分隔的（CSV）文本文件：

```
In [8]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

> 笔记：这里，我用的是Unix的cat shell命令将文件的原始内容打印到屏幕上。如果你用的是Windows，你可以使用type达到同样的效果。

由于该文件以逗号分隔，所以我们可以使用read_csv将其读入一个DataFrame：

```
In [9]: df = pd.read_csv('examples/ex1.csv')

In [10]: df
Out[10]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

我们还可以使用read_table，并指定分隔符：

```
In [11]: pd.read_table('examples/ex1.csv', sep=',')
Out[11]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

并不是所有文件都有标题行。看看下面这个文件：

```
In [12]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

读入该文件的办法有两个。你可以让pandas为其分配默认的列名，也可以自己定义列名：

```
In [13]: pd.read_csv('examples/ex2.csv', header=None)
Out[13]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo

In [14]: pd.read_csv('examples/ex2.csv', names=['a',
Out[14]:
```

```
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

假设你希望将message列做成DataFrame的索引。你可以明确表示要将该列放到索引4的位置上，也可以通过index_col参数指定"message"：

```
In [15]: names = ['a', 'b', 'c', 'd', 'message']

In [16]: pd.read_csv('examples/ex2.csv', names=names,
Out[16]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

如果希望将多个列做成一个层次化索引，只需传入由列编号或列名组成的列表即可：

```
In [17]: !cat examples/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [18]: parsed = pd.read_csv('examples/csv_mindex.cs
   ....:                      index_col=['key1', 'key
```

```
In [19]: parsed
Out[19]:
           value1  value2
key1 key2
one  a          1       2
     b          3       4
     c          5       6
     d          7       8
two  a          9      10
     b         11      12
     c         13      14
     d         15      16
```

有些情况下，有些表格可能不是用固定的分隔符去分隔字段的（比如空白符或其他模式）。有些表格可能不是用固定的分隔符去分隔字段的（比如空白符或其他模式来分隔字段）。看看下面这个文本文件：

```
In [20]: list(open('examples/ex3.txt'))
Out[20]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']
```

虽然可以手动对数据进行规整，这里的字段是被数量不同的空白字符间隔开的。这种情况下，你可以传递一个正则表达式作为read_table的分隔符。可以用正则表达式表达为\s+，于是有有：

```
In [21]: result = pd.read_table('examples/ex3.txt', s

In [22]: result
Out[22]:
            A         B         C
```

```
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

这里，由于列名比数据行的数量少，所以read_table推断第一列应该是DataFrame的索引。这里，由于列名比数据行的数量少，所以read_table推断第一列应该是DataFrame的索引。

这些解析器函数还有许多参数可以帮助你处理各种各样的异形文件格式（表6-2列出了一些）。比如说，你可以用skiprows跳过文件的第一行、第三行和第四行：

```
In [23]: !cat examples/ex4.csv
# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
In [24]: pd.read_csv('examples/ex4.csv', skiprows=[0,
Out[24]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

缺失值处理是文件解析任务中的一个重要组成部分。缺失数据经常是要么没有（空字符串），要么用某个标记值表示。默认情况下，pandas会用一组经常出现的标记值进行识别，比如NA及NULL：

```
In [25]: !cat examples/ex5.csv
something,a,b,c,d,message
```

```
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo
In [26]: result = pd.read_csv('examples/ex5.csv')

In [27]: result
Out[27]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo

In [28]: pd.isnull(result)
Out[28]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

na_values可以用一个列表或集合的字符串表示缺失值：

```
In [29]: result = pd.read_csv('examples/ex5.csv', na_

In [30]: result
Out[30]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

字典的各列可以使用不同的NA标记值：

```
In [31]: sentinels = {'message': ['foo', 'NA'], 'some

In [32]: pd.read_csv('examples/ex5.csv', na_values=se
Out[32]:
```

```
something    a   b     c   d message
0          one   1   2   3.0   4     NaN
1          NaN   5   6   NaN   8   world
2        three   9  10  11.0  12     NaN
```

表6-2列出了pandas.read_csv和pandas.read_table常用的选项。

表6-2：read_csv/read_table函数的参数

参数	说明
path	表示文件系统位置、URL、文件型对象的字符串
sep或delimiter	用于对行中各字段进行拆分的字符序列或正则表达式
header	用作列名的行号。默认为0（第一行），如果没有header行就应该设置为None
index_col	用作行索引的列编号或列名。可以是单个名称/数字或由多个名称/数字组成的列表（层次化索引）
names	用于结果的列名列表，结合header=None
skiprows	需要忽略的行数（从文件开始处算起），或需要跳过的行号列表（从0开始）
na_values	一组用于替换NA的值
comment	用于将注释信息从行尾拆分出去的字符（一个或多个）
parse_dates	尝试将数据解析为日期，默认为False。如果为True，则尝试解析所有列。此外，还可以指定需要解析的一组列号或列名。如果列表的元素为列表或元组，就会将多个列组合到一起再进行日期解析工作（例如，日期/时间分别位于两个列中）
keep_date_col	如果连接多列解析日期，则保持参与连接的列。默认为False。
converters	由列号/列名跟函数之间的映射关系组成的字典。例如，{'foo': f}会对foo列的所有值应用函数f
dayfirst	当解析有歧义的日期时，将其看做国际格式（例如，7/6/2012 → June 7, 2012）。默认为False
date_parser	用于解析日期的函数
nrows	需要读取的行数（从文件开始处算起）
iterator	返回一个TextParser以便逐块读取文件
chunksize	文件块的大小（用于迭代）
skip_footer	需要忽略的行数（从文件末尾处算起）

表6-2：read_csv/read_table函数的参数（续）

参数	说明
verbose	打印各种解析器输出信息，比如“非数值列中缺失值的数量”等
encoding	用于unicode的文本编码格式。例如，“utf-8”表示用UTF-8编码的文本
squeeze	如果数据经解析后仅含一列，则返回Series
thousands	千分位分隔符，如“,”或“.”

逐块读取文本文件

在处理很大的文件时，或找出大文件中的参数集以便于后续处理时，你可能只想读取文件的一小部分或逐块对文件进行迭代。

在看大文件之前，我们先设置pandas显示地更紧些：

```
In [33]: pd.options.display.max_rows = 10
```

然后有：

```
In [34]: result = pd.read_csv('examples/ex6.csv')

In [35]: result
Out[35]:
           one       two     three      four key
0     0.467976 -0.038649 -0.295344 -1.824726   L
1    -0.358893  1.404453  0.704965 -0.200638   B
2    -0.501840  0.659254 -0.421691 -0.057688   G
3     0.204886  1.074134  1.388361 -0.982404   R
4     0.354628 -0.133116  0.283763 -0.837063   Q
...        ...       ...       ...       ...  ..
9995  2.311896 -0.417070 -1.409599 -0.515821   L
9996 -0.479893 -0.650419  0.745152 -0.646038   E
9997  0.523331  0.787112  0.486066  1.093156   K
9998 -0.362559  0.598894 -1.843201  0.887292   G
9999 -0.096376 -1.012999 -0.657431 -0.573315   0
[10000 rows x 5 columns]
If you want to only read a small
```

如果只想读取几行（避免读取整个文件），通过nrows进行指定即可：

```
In [36]: pd.read_csv('examples/ex6.csv', nrows=5)
Out[36]:
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

要逐块读取文件，可以指定chunksize（行数）：

```
In [874]: chunker = pd.read_csv('ch06/ex6.csv', chunk

In [875]: chunker
Out[875]: <pandas.io.parsers.TextParser at 0x8398150>
```

read_csv所返回的这个TextParser对象使你可以根据chunksize对文件进行逐块迭代。比如说，我们可以迭代处理ex6.csv，将值计数聚合到"key"列中，如下所示：

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1

tot = pd.Series([])
for piece in chunker:
    tot = tot.add(piece['key'].value_counts(), fill_v

tot = tot.sort_values(ascending=False)
```

然后有：

```
In [40]: tot[:10]
Out[40]:
```

```
E    368.0
X    364.0
L    346.0
O    343.0
Q    340.0
M    338.0
J    337.0
F    335.0
K    334.0
H    330.0
dtype: float64
```

TextParser还有一个get_chunk方法，它使你可以读取任意大小的块。

将数据写出到文本格式

数据也可以被输出为分隔符格式的文本。我们再来看看之前读过的一个CSV文件：

```
In [41]: data = pd.read_csv('examples/ex5.csv')

In [42]: data
Out[42]:
  something  a   b     c   d message
0       one  1   2   3.0   4     NaN
1       two  5   6   NaN   8   world
2     three  9  10  11.0  12     foo
```

利用DataFrame的to_csv方法，我们可以将数据写到一个以逗号分隔的文件中：

```
In [43]: data.to_csv('examples/out.csv')

In [44]: !cat examples/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
```

```
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

当然，还可以使用其他分隔符（由于这里直接写出到sys.stdout，所以仅仅是打印出文本结果而已）：

```
In [45]: import sys

In [46]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

缺失值在输出结果中会被表示为空字符串。你可能希望将其表示为别的标记值：

```
In [47]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

如果没有设置其他选项，则会写出行和列的标签。当然，它们也都可以被禁用：

```
In [48]: data.to_csv(sys.stdout, index=False, header=
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

此外，你还可以只写出一部分的列，并以你指定的顺序排列：

```
In [49]: data.to_csv(sys.stdout, index=False, columns=
a,b,c
1,2,3.0
```

```
5,6,
9,10,11.0
```

Series也有一个to_csv方法：

```
In [50]: dates = pd.date_range('1/1/2000', periods=7)

In [51]: ts = pd.Series(np.arange(7), index=dates)

In [52]: ts.to_csv('examples/tseries.csv')

In [53]: !cat examples/tseries.csv
2000-01-01,0
2000-01-02,1
2000-01-03,2
2000-01-04,3
2000-01-05,4
2000-01-06,5
2000-01-07,6
```

处理分隔符格式

大部分存储在磁盘上的表格型数据都能用pandas.read_table进行加载。然而，有时还是需要做一些手工处理。由于接收到含有畸形行的文件而使read_table出毛病的情况并不少见。为了说明这些基本工具，看看下面这个简单的CSV文件：

```
In [54]: !cat examples/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3"
```

对于任何单字符分隔符文件，可以直接使用Python内置的csv模块。将任意已打开的文件或文件型的对象传给csv.reader：

```
import csv
f = open('examples/ex7.csv')

reader = csv.reader(f)
```

对这个reader进行迭代将会为每行产生一个元组（并移除了所有的引号）：对这个reader进行迭代将会为每行产生一个元组（并移除了所有的引号）：

```
In [56]: for line in reader:
   ....:     print(line)
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3']
```

现在，为了使数据格式合乎要求，你需要对其做一些整理工作。我们一步一步来做。首先，读取文件到一个多行的列表中：

```
In [57]: with open('examples/ex7.csv') as f:
   ....:     lines = list(csv.reader(f))
```

然后，我们将这些行分为标题行和数据行：

```
In [58]: header, values = lines[0], lines[1:]
```

然后，我们可以用字典构造式和zip(*values)，后者将行转置为列，创建数据列的字典：

```
In [59]: data_dict = {h: v for h, v in zip(header, zi

In [60]: data_dict
Out[60]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3'
```

CSV文件的形式有很多。只需定义csv.Dialect的一个子类即可定义出新格式（如专门的分隔符、字符串引用约定、行结束符

等）：

```
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL
reader = csv.reader(f, dialect=my_dialect)
```

各个CSV语支的参数也可以关键字的形式提供给csv.reader，而无需定义子类：

```
reader = csv.reader(f, delimiter='|')
```

可用的选项（csv.Dialect的属性）及其功能如表6-3所示。

表6-3：CSV语支选项

参数	说明
delimiter	用于分隔字段的单字符字符串。默认为“,”
lineterminator	用于写操作的行结束符，默认为“\r\n”。读操作将忽略此选项，它能认出跨平台的行结束符
quotechar	用于带有特殊字符（如分隔符）的字段的引用符号。默认为“"”
quoting	引用约定。可选值包括csv.QUOTE_ALL（引用所有字段）、csv.QUOTE_MINIMAL（只引用带有诸如分隔符之类特殊字符的字段）、csv.QUOTE_NONNUMERIC以及csv.QUOTE_NON（不引用）。完整信息请参考Python的文档。默认为QUOTE_MINIMAL
skipinitialspace	忽略分隔符后面的空白符。默认为False
doublequote	如何处理字段内的引用符号。如果为True，则双写。完整信息及行为请参见在线文档
escapechar	用于对分隔符进行转义的字符串（如果quoting被设置为csv.QUOTE_NONE的话）。默认禁用

> 笔记：对于那些使用复杂分隔符或多字符分隔符的文件，csv模块就无能为力了。这种情况下，你就只能使用字符串的split方法或正则表达式方法re.split进行行拆分和其他整理工作了。

要手工输出分隔符文件，你可以使用csv.writer。它接受一个已打

开且可写的文件对象以及跟csv.reader相同的那些语支和格式化选项：

```
with open('mydata.csv', 'w') as f:
    writer = csv.writer(f, dialect=my_dialect)
    writer.writerow(('one', 'two', 'three'))
    writer.writerow(('1', '2', '3'))
    writer.writerow(('4', '5', '6'))
    writer.writerow(('7', '8', '9'))
```

JSON数据

JSON（JavaScript Object Notation的简称）已经成为通过HTTP请求在Web浏览器和其他应用程序之间发送数据的标准格式之一。它是一种比表格型文本格式（如CSV）灵活得多的数据格式。下面是一个例子：

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"
 "pet": null,
 "siblings": [{"name": "Scott", "age": 30, "pets": ["
              {"name": "Katie", "age": 38,
               "pets": ["Sixes", "Stache", "Cisco"]}]
}
"""

除其空值null和一些其他的细微差别（如列表末尾不允许存在多余的逗号
```python
In [62]: import json

In [63]: result = json.loads(obj)

In [64]: result
Out[64]:
{'name': 'Wes',
 'pet': None,
```
```

```
 'places_lived': ['United States', 'Spain', 'Germany'
 'siblings': [{'age': 30, 'name': 'Scott', 'pets': ['
  {'age': 38, 'name': 'Katie', 'pets': ['Sixes', 'Sta
```

json.dumps则将Python对象转换成JSON格式：

```
In [65]: asjson = json.dumps(result)
```

如何将（一个或一组）JSON对象转换为DataFrame或其他便于分析的数据结构就由你决定了。最简单方便的方式是：向DataFrame构造器传入一个字典的列表（就是原先的JSON对象），并选取数据字段的子集：

```
In [66]: siblings = pd.DataFrame(result['siblings'], 

In [67]: siblings
Out[67]: 
    name  age
0  Scott   30
1  Katie   38
```

pandas.read_json可以自动将特别格式的JSON数据集转换为Series或DataFrame。例如：

```
In [68]: !cat examples/example.json
[{"a": 1, "b": 2, "c": 3},
 {"a": 4, "b": 5, "c": 6},
 {"a": 7, "b": 8, "c": 9}]
```

pandas.read_json的默认选项假设JSON数组中的每个对象是表格中的一行：

```
In [69]: data = pd.read_json('examples/example.json')

In [70]: data
```

```
Out[70]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

第7章中关于USDA Food Database的那个例子进一步讲解了JSON数据的读取和处理（包括嵌套记录）。

如果你需要将数据从pandas输出到JSON，可以使用to_json方法：

```
In [71]: print(data.to_json())
{"a":{"0":1,"1":4,"2":7},"b":{"0":2,"1":5,"2":8},"c":

In [72]: print(data.to_json(orient='records'))
[{"a":1,"b":2,"c":3},{"a":4,"b":5,"c":6},{"a":7,"b":8
```

XML和HTML：Web信息收集

Python有许多可以读写常见的HTML和XML格式数据的库，包括lxml、Beautiful Soup和html5lib。lxml的速度比较快，但其它的库处理有误的HTML或XML文件更好。

pandas有一个内置的功能，read_html，它可以使用lxml和Beautiful Soup自动将HTML文件中的表格解析为DataFrame对象。为了进行展示，我从美国联邦存款保险公司下载了一个HTML文件（pandas文档中也使用过），它记录了银行倒闭的情况。首先，你需要安装read_html用到的库：

```
conda install lxml
pip install beautifulsoup4 html5lib
```

如果你用的不是conda，可以使用`pip install lxml`。

pandas.read_html有一些选项，默认条件下，它会搜索、尝试解

析<table>标签内的的表格数据。结果是一个列表的DataFrame对象：

```
In [73]: tables = pd.read_html('examples/fdic_failed_

In [74]: len(tables)
Out[74]: 1

In [75]: failures = tables[0]

In [76]: failures.head()
Out[76]:
                      Bank Name             City  ST
0                   Allied Bank         Mulberry  AR
1  The Woodbury Banking Company         Woodbury  GA
2        First CornerStone Bank  King of Prussia  PA
3            Trust Company Bank          Memphis  TN
4    North Milwaukee State Bank        Milwaukee  WI
                 Acquiring Institution        Closing
0                          Today's Bank  September 23,
1                           United Bank     August 19,
2  First-Citizens Bank & Trust Company         May 6,
3           The Bank of Fayette County       April 29,
4  First-Citizens Bank & Trust Company       March 11,
```

因为failures有许多列，pandas插入了一个换行符\。

这里，我们可以做一些数据清洗和分析（后面章节会进一步讲解），比如计算按年份计算倒闭的银行数：

```
In [77]: close_timestamps = pd.to_datetime(failures['

In [78]: close_timestamps.dt.year.value_counts()
Out[78]:
2010    157
2009    140
```

```
2011     92
2012     51
2008     25
       ...
2004      4
2001      4
2007      3
2003      3
2000      2
Name: Closing Date, Length: 15, dtype: int64
```

利用lxml.objectify解析XML

XML（Extensible Markup Language）是另一种常见的支持分层、嵌套数据以及元数据的结构化数据格式。本书所使用的这些文件实际上来自于一个很大的XML文档。

前面，我介绍了pandas.read_html函数，它可以使用lxml或Beautiful Soup从HTML解析数据。XML和HTML的结构很相似，danXML更为通用。这里，我会用一个例子演示如何利用lxml从XML格式解析数据。

纽约大都会运输署发布了一些有关其公交和列车服务的数据资料（http://www.mta.info/developers/download.html）。这里，我们将看看包含在一组XML文件中的运行情况数据。每项列车或公交服务都有各自的文件（如Metro-North Railroad的文件是Performance_MNR.xml），其中每条XML记录就是一条月度数据，如下所示：

```
<INDICATOR>
  <INDICATOR_SEQ>373889</INDICATOR_SEQ>
  <PARENT_SEQ></PARENT_SEQ>
  <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
  <INDICATOR_NAME>Escalator Availability</INDICATOR_N
  <DESCRIPTION>Percent of the time that escalators ar
  systemwide. The availability rate is based on physi
```

```
  the morning of regular business days only. This is a
  began reporting in 2009.</DESCRIPTION>
  <PERIOD_YEAR>2011</PERIOD_YEAR>
  <PERIOD_MONTH>12</PERIOD_MONTH>
  <CATEGORY>Service Indicators</CATEGORY>
  <FREQUENCY>M</FREQUENCY>
  <DESIRED_CHANGE>U</DESIRED_CHANGE>
  <INDICATOR_UNIT>%</INDICATOR_UNIT>
  <DECIMAL_PLACES>1</DECIMAL_PLACES>
  <YTD_TARGET>97.00</YTD_TARGET>
  <YTD_ACTUAL></YTD_ACTUAL>
  <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
  <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

我们先用lxml.objectify解析该文件，然后通过getroot得到该XML文件的根节点的引用：

```
from lxml import objectify

path = 'examples/mta_perf/Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

root.INDICATOR返回一个用于产生各个<INDICATOR>XML元素的生成器。对于每条记录，我们可以用标记名（如YTD_ACTUAL）和数据值填充一个字典（排除几个标记）：

```
data = []

skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'DESIRED_CHANGE', 'DECIMAL_PLACES']

for elt in root.INDICATOR:
    el_data = {}
    for child in elt.getchildren():
```

```
        if child.tag in skip_fields:
            continue
        el_data[child.tag] = child.pyval
    data.append(el_data)
```

最后，将这组字典转换为一个DataFrame：

```
In [81]: perf = pd.DataFrame(data)

In [82]: perf.head()
Out[82]:
Empty DataFrame
Columns: []
Index: []
```

XML数据可以比本例复杂得多。每个标记都可以有元数据。看看下面这个HTML的链接标签（它也算是一段有效的XML）：

```
from io import StringIO
tag = '<a href="http://www.google.com">Google</a>'
root = objectify.parse(StringIO(tag)).getroot()
```

现在就可以访问标签或链接文本中的任何字段了（如href）：

```
In [84]: root
Out[84]: <Element a at 0x7f6b15817748>

In [85]: root.get('href')
Out[85]: 'http://www.google.com'

In [86]: root.text
Out[86]: 'Google'
```

6.2 二进制数据格式

实现数据的高效二进制格式存储最简单的办法之一是使用Python内置的pickle序列化。pandas对象都有一个用于将数据以pickle

格式保存到磁盘上的to_pickle方法：

```
In [87]: frame = pd.read_csv('examples/ex1.csv')

In [88]: frame
Out[88]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo

In [89]: frame.to_pickle('examples/frame_pickle')
```

你可以通过pickle直接读取被pickle化的数据，或是使用更为方便的pandas.read_pickle：

```
In [90]: pd.read_pickle('examples/frame_pickle')
Out[90]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

> 注意：pickle仅建议用于短期存储格式。其原因是很难保证该格式永远是稳定的；今天pickle的对象可能无法被后续版本的库unpickle出来。虽然我尽力保证这种事情不会发生在pandas中，但是今后的某个时候说不定还是得“打破”该pickle格式。

pandas内置支持两个二进制数据格式：HDF5和MessagePack。下一节，我会给出几个HDF5的例子，但我建议你尝试下不同的文件格式，看看它们的速度以及是否适合你的分析工作。pandas或NumPy数据的其它存储格式有：

- bcolz：一种可压缩的列存储二进制格式，基于Blosc压缩库。
- Feather：我与R语言社区的Hadley Wickham设计的一种跨语

言的列存储文件格式。Feather使用了Apache Arrow的列式内存格式。

使用HDF5格式

HDF5是一种存储大规模科学数组数据的非常好的文件格式。它可以被作为C库，带有许多语言的接口，如Java、Python和MATLAB等。HDF5中的HDF指的是层次型数据格式（hierarchical data format）。每个HDF5文件都含有一个文件系统式的节点结构，它使你能够存储多个数据集并支持元数据。与其他简单格式相比，HDF5支持多种压缩器的即时压缩，还能更高效地存储重复模式数据。对于那些非常大的无法直接放入内存的数据集，HDF5就是不错的选择，因为它可以高效地分块读写。

虽然可以用PyTables或h5py库直接访问HDF5文件，pandas提供了更为高级的接口，可以简化存储Series和DataFrame对象。HDFStore类可以像字典一样，处理低级的细节：

```
In [92]: frame = pd.DataFrame({'a': np.random.randn(1

In [93]: store = pd.HDFStore('mydata.h5')

In [94]: store['obj1'] = frame

In [95]: store['obj1_col'] = frame['a']

In [96]: store
Out[96]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
/obj1                frame        (shape->[100,1])

/obj1_col            series       (shape->[100])

/obj2                frame_table  (typ->appendable,nr
```

```
[index])
/obj3                frame_table  (typ->appendable,nr
[index])
```

HDF5文件中的对象可以通过与字典一样的API进行获取：

```
In [97]: store['obj1']
Out[97]:
           a
0  -0.204708
1   0.478943
2  -0.519439
3  -0.555730
4   1.965781
..       ...
95  0.795253
96  0.118110
97 -0.748532
98  0.584970
99  0.152677
[100 rows x 1 columns]
```

HDFStore支持两种存储模式，'fixed'和'table'。后者通常会更慢，但是支持使用特殊语法进行查询操作：

```
In [98]: store.put('obj2', frame, format='table')

In [99]: store.select('obj2', where=['index >= 10 and
Out[99]:
           a
10  1.007189
11 -1.296221
12  0.274992
13  0.228913
14  1.352917
15  0.886429
```

```
In [100]: store.close()
```

put是store['obj2'] = frame方法的显示版本，允许我们设置其它的选项，比如格式。

pandas.read_hdf函数可以快捷使用这些工具：

```
In [101]: frame.to_hdf('mydata.h5', 'obj3', format='ta

In [102]: pd.read_hdf('mydata.h5', 'obj3', where=['inc
Out[102]:
          a
0 -0.204708
1  0.478943
2 -0.519439
3 -0.555730
4  1.965781
```

> 笔记：如果你要处理的数据位于远程服务器，比如Amazon S3或HDFS，使用专门为分布式存储（比如Apache Parquet）的二进制格式也许更加合适。Python的Parquet和其它存储格式还在不断的发展之中，所以这本书中没有涉及。

如果需要本地处理海量数据，我建议你好好研究一下PyTables和h5py，看看它们能满足你的哪些需求。。由于许多数据分析问题都是IO密集型（而不是CPU密集型），利用HDF5这样的工具能显著提升应用程序的效率。

> 注意：HDF5不是数据库。它最适合用作“一次写多次读”的数据集。虽然数据可以在任何时候被添加到文件中，但如果同时发生多个写操作，文件就可能会被破坏。

读取Microsoft Excel文件

pandas的ExcelFile类或pandas.read_excel函数支持读取存储在Excel 2003（或更高版本）中的表格型数据。这两个工具分别使用扩展包xlrd和openpyxl读取XLS和XLSX文件。你可以用pip或conda安装它们。

要使用ExcelFile，通过传递xls或xlsx路径创建一个实例：

```
In [104]: xlsx = pd.ExcelFile('examples/ex1.xlsx')
```

存储在表单中的数据可以read_excel读取到DataFrame（原书这里写的是用parse解析，但代码中用的是read_excel，是个笔误：只换了代码，没有改文字）：

```
In [105]: pd.read_excel(xlsx, 'Sheet1')
Out[105]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

如果要读取一个文件中的多个表单，创建ExcelFile会更快，但你也可以将文件名传递到pandas.read_excel：

```
In [106]: frame = pd.read_excel('examples/ex1.xlsx',

In [107]: frame
Out[107]:
   a   b   c   d message
0  1   2   3   4   hello
1  5   6   7   8   world
2  9  10  11  12     foo
```

如果要将pandas数据写入为Excel格式，你必须首先创建一个ExcelWriter，然后使用pandas对象的to_excel方法将数据写入到其中：

```
In [108]: writer = pd.ExcelWriter('examples/ex2.xlsx'

In [109]: frame.to_excel(writer, 'Sheet1')

In [110]: writer.save()
```

你还可以不使用ExcelWriter，而是传递文件的路径到to_excel：

```
In [111]: frame.to_excel('examples/ex2.xlsx')
```

6.3 Web APIs交互

许多网站都有一些通过JSON或其他格式提供数据的公共API。通过Python访问这些API的办法有不少。一个简单易用的办法（推荐）是requests包（http://docs.python-requests.org）。

为了搜索最新的30个GitHub上的pandas主题，我们可以发一个HTTP GET请求，使用requests扩展库：

```
In [113]: import requests

In [114]: url = 'https://api.github.com/repos/pandas-

In [115]: resp = requests.get(url)

In [116]: resp
Out[116]: <Response [200]>
```

响应对象的json方法会返回一个包含被解析过的JSON字典，加载到一个Python对象中：

```
In [117]: data = resp.json()

In [118]: data[0]['title']
Out[118]: 'Period does not round down for frequencies
```

data中的每个元素都是一个包含所有GitHub主题页数据（不包含评论）的字典。我们可以直接传递数据到DataFrame，并提取感兴趣的字段：

```
In [119]: issues = pd.DataFrame(data, columns=['numbe
   .....:                                        'label

In [120]: issues
Out[120]:
    number
0    17666  Period does not round down for frequencie
1    17665           DOC: improve docstring of functi
2    17664               COMPAT: skip 32-bit test on 
3    17662                          implement Delegat
4    17654  BUG: Fix series rename called with str al
..     ...
25   17603  BUG: Correctly localize naive datetime st
26   17599                     core.dtypes.generic --
27   17596   Merge cdate_range functionality into bda
28   17587  Time Grouper bug fix when applied for lis
29   17583  BUG: fix tz-aware DatetimeIndex + Timedel
                                               labels
0                                                  []
1   [{'id': 134699, 'url': 'https://api.github.com...
2   [{'id': 563047854, 'url': 'https://api.github....
3                                                  []
4   [{'id': 76811, 'url': 'https://api.github.com/...
..                                                ...
25  [{'id': 76811, 'url': 'https://api.github.com/...
26  [{'id': 49094459, 'url': 'https://api.github.c...
27  [{'id': 35818298, 'url': 'https://api.github.c...
28  [{'id': 233160, 'url': 'https://api.github.com...
29  [{'id': 76811, 'url': 'https://api.github.com/...
[30 rows x 4 columns]
```

花费一些精力，你就可以创建一些更高级的常见的Web API的接口，返回DataFrame对象，方便进行分析。

6.4 数据库交互

在商业场景下，大多数数据可能不是存储在文本或Excel文件中。基于SQL的关系型数据库（如SQL Server、PostgreSQL和MySQL等）使用非常广泛，其它一些数据库也很流行。数据库的选择通常取决于性能、数据完整性以及应用程序的伸缩性需求。

将数据从SQL加载到DataFrame的过程很简单，此外pandas还有一些能够简化该过程的函数。例如，我将使用SQLite数据库（通过Python内置的sqlite3驱动器）：

```
In [121]: import sqlite3

In [122]: query = """
   .....: CREATE TABLE test
   .....: (a VARCHAR(20), b VARCHAR(20),
   .....:  c REAL,        d INTEGER
   .....: );"""

In [123]: con = sqlite3.connect('mydata.sqlite')

In [124]: con.execute(query)
Out[124]: <sqlite3.Cursor at 0x7f6b12a50f10>

In [125]: con.commit()
```

然后插入几行数据：

```
In [126]: data = [('Atlanta', 'Georgia', 1.25, 6),
   .....:         ('Tallahassee', 'Florida', 2.6, 3),
   .....:         ('Sacramento', 'California', 1.7, 5

In [127]: stmt = "INSERT INTO test VALUES(?, ?, ?, ?)
```

```
In [128]: con.executemany(stmt, data)
Out[128]: <sqlite3.Cursor at 0x7f6b15c66ce0>
```

从表中选取数据时，大部分Python SQL驱动器（PyODBC、psycopg2、MySQLdb、pymssql等）都会返回一个元组列表：

```
In [130]: cursor = con.execute('select * from test')

In [131]: rows = cursor.fetchall()

In [132]: rows
Out[132]:
[('Atlanta', 'Georgia', 1.25, 6),
 ('Tallahassee', 'Florida', 2.6, 3),
 ('Sacramento', 'California', 1.7, 5)]
```

你可以将这个元组列表传给DataFrame构造器，但还需要列名（位于光标的description属性中）：

```
In [133]: cursor.description
Out[133]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [134]: pd.DataFrame(rows, columns=[x[0] for x in c
Out[134]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

这种数据规整操作相当多，你肯定不想每查一次数据库就重写一

次。SQLAlchemy项目是一个流行的Python SQL工具，它抽象出了SQL数据库中的许多常见差异。pandas有一个read_sql函数，可以让你轻松的从SQLAlchemy连接读取数据。这里，我们用SQLAlchemy连接SQLite数据库，并从之前创建的表读取数据：

```
In [135]: import sqlalchemy as sqla

In [136]: db = sqla.create_engine('sqlite:///mydata.s

In [137]: pd.read_sql('select * from test', db)
Out[137]:
             a           b     c  d
0      Atlanta     Georgia  1.25  6
1  Tallahassee     Florida  2.60  3
2   Sacramento  California  1.70  5
```

6.5 总结

访问数据通常是数据分析的第一步。在本章中，我们已经学了一些有用的工具。在接下来的章节中，我们将深入研究数据规整、数据可视化、时间序列分析和其它主题。

第7章 数据清洗和准备

在数据分析和建模的过程中，相当多的时间要用在数据准备上：加载、清理、转换以及重塑。这些工作会占到分析师时间的80%或更多。有时，存储在文件和数据库中的数据的格式不适合某个特定的任务。许多研究者都选择使用通用编程语言（如Python、Perl、R或Java）或UNIX文本处理工具（如sed或awk）对数据格式进行专门处理。幸运的是，pandas和内置的Python标准库提供了一组高级的、灵活的、快速的工具，可以让你轻松地将数据规变为想要的格式。

如果你发现了一种本书或pandas库中没有的数据操作方式，请尽管在邮件列表或GitHub网站上提出。实际上，pandas的许多设计和实现都是由真实应用的需求所驱动的。

在本章中，我会讨论处理缺失数据、重复数据、字符串操作和其它分析数据转换的工具。下一章，我会关注于用多种方法合并、重塑数据集。

7.1 处理缺失数据

在许多数据分析工作中，缺失数据是经常发生的。pandas的目标之一就是尽量轻松地处理缺失数据。例如，pandas对象的所有描述性统计默认都不包括缺失数据。

缺失数据在pandas中呈现的方式有些不完美，但对于大多数用户可以保证功能正常。对于数值数据，pandas使用浮点值NaN（Not a Number）表示缺失数据。我们称其为哨兵值，可以方便的检测出来：

```
In [10]: string_data = pd.Series(['aardvark', 'artich

In [11]: string_data
Out[11]:
0     aardvark
1    artichoke
```

```
2          NaN
3      avocado
dtype: object

In [12]: string_data.isnull()
Out[12]:
0    False
1    False
2     True
3    False
dtype: bool
```

在pandas中，我们采用了R语言中的惯用法，即将缺失值表示为NA，它表示不可用not available。在统计应用中，NA数据可能是不存在的数据或者虽然存在，但是没有观察到（例如，数据采集中发生了问题）。当进行数据清洗以进行分析时，最好直接对缺失数据进行分析，以判断数据采集的问题或缺失数据可能导致的偏差。

Python内置的None值在对象数组中也可以作为NA：

```
In [13]: string_data[0] = None

In [14]: string_data.isnull()
Out[14]:
0     True
1    False
2     True
3    False
dtype: bool
```

pandas项目中还在不断优化内部细节以更好处理缺失数据，像用户API功能，例如pandas.isnull，去除了许多恼人的细节。表7-1列出了一些关于缺失数据处理的函数。

方法	说明
dropna	根据各标签的值中是否存在缺失数据对轴标签进行过滤，可通过阈值调节对缺失值的容忍度
fillna	用指定值或插值方法（如ffill或bfill）填充缺失数据
isnull	返回一个含有布尔值的对象，这些布尔值表示哪些值是缺失值/NA，该对象的类型与源类型一样
notnull	isnull的否定式

表7-1 NA处理方法

滤除缺失数据

过滤掉缺失数据的办法有很多种。你可以通过pandas.isnull或布尔索引的手工方法，但dropna可能会更实用一些。对于一个Series，dropna返回一个仅含非空数据和索引值的Series：

```
In [15]: from numpy import nan as NA

In [16]: data = pd.Series([1, NA, 3.5, NA, 7])

In [17]: data.dropna()
Out[17]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

这等价于：

```
In [18]: data[data.notnull()]
Out[18]:
0    1.0
2    3.5
4    7.0
dtype: float64
```

而对于DataFrame对象，事情就有点复杂了。你可能希望丢弃全NA或含有NA的行或列。dropna默认丢弃任何含有缺失值的行：

```
In [19]: data = pd.DataFrame([[1., 6.5, 3.], [1., NA,
   ....:                      [NA, NA, NA], [NA, 6.5,

In [20]: cleaned = data.dropna()

In [21]: data
Out[21]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0

In [22]: cleaned
Out[22]:
     0    1    2
0  1.0  6.5  3.0
```

传入how='all'将只丢弃全为NA的那些行：

```
In [23]: data.dropna(how='all')
Out[23]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

用这种方式丢弃列，只需传入axis=1即可：

```
In [24]: data[4] = NA

In [25]: data
Out[25]:
     0    1    2   4
0  1.0  6.5  3.0 NaN
1  1.0  NaN  NaN NaN
```

```
2  NaN  NaN  NaN NaN
3  NaN  6.5  3.0 NaN

In [26]: data.dropna(axis=1, how='all')
Out[26]:
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
```

另一个滤除DataFrame行的问题涉及时间序列数据。假设你只想留下一部分观测数据，可以用thresh参数实现此目的：

```
In [27]: df = pd.DataFrame(np.random.randn(7, 3))

In [28]: df.iloc[:4, 1] = NA

In [29]: df.iloc[:2, 2] = NA

In [30]: df
Out[30]:
          0         1         2
0 -0.204708       NaN       NaN
1 -0.555730       NaN       NaN
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741

In [31]: df.dropna()
Out[31]:
          0         1         2
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
In [32]: df.dropna(thresh=2)
Out[32]:
          0         1         2
2  0.092908       NaN  0.769023
3  1.246435       NaN -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

填充缺失数据

你可能不想滤除缺失数据（有可能会丢弃跟它有关的其他数据），而是希望通过其他方式填补那些“空洞”。对于大多数情况而言，fillna方法是最主要的函数。通过一个常数调用fillna就会将缺失值替换为那个常数值：

```
In [33]: df.fillna(0)
Out[33]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

若是通过一个字典调用fillna，就可以实现对不同的列填充不同的值：

```
In [34]: df.fillna({1: 0.5, 2: 0})
Out[34]:
          0         1         2
0 -0.204708  0.500000  0.000000
1 -0.555730  0.500000  0.000000
2  0.092908  0.500000  0.769023
```

```
3  1.246435  0.500000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

fillna默认会返回新对象，但也可以对现有对象进行就地修改：

```
In [35]: _ = df.fillna(0, inplace=True)

In [36]: df
Out[36]:
          0         1         2
0 -0.204708  0.000000  0.000000
1 -0.555730  0.000000  0.000000
2  0.092908  0.000000  0.769023
3  1.246435  0.000000 -1.296221
4  0.274992  0.228913  1.352917
5  0.886429 -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

对reindexing有效的那些插值方法也可用于fillna：

```
In [37]: df = pd.DataFrame(np.random.randn(6, 3))

In [38]: df.iloc[2:, 1] = NA

In [39]: df.iloc[4:, 2] = NA

In [40]: df
Out[40]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772       NaN  1.343810
3 -0.713544       NaN -2.370232
4 -1.860761       NaN       NaN
5 -1.265934       NaN       NaN
```

```
In [41]: df.fillna(method='ffill')
Out[41]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761  0.124121 -2.370232
5 -1.265934  0.124121 -2.370232

In [42]: df.fillna(method='ffill', limit=2)
Out[42]:
          0         1         2
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
2  0.523772  0.124121  1.343810
3 -0.713544  0.124121 -2.370232
4 -1.860761       NaN -2.370232
5 -1.265934       NaN -2.370232
```

只要有些创新，你就可以利用fillna实现许多别的功能。比如说，你可以传入Series的平均值或中位数：

```
In [43]: data = pd.Series([1., NA, 3.5, NA, 7])

In [44]: data.fillna(data.mean())
Out[44]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
dtype: float64
```

表7-2列出了fillna的参考。

参数	说明
value	用于填充缺失值的标量值或字典对象
method	插值方式。如果函数调用时未指定其他参数的话，默认为“ffill”
axis	待填充的轴，默认axis=0
inplace	修改调用者对象而不产生副本
limit	（对于前向和后向填充）可以连续填充的最大数量

fillna函数参数

7.2 数据转换

本章到目前为止介绍的都是数据的重排。另一类重要操作则是过滤、清理以及其他的转换工作。

移除重复数据

DataFrame中出现重复行有多种原因。下面就是一个例子：

```
In [45]: data = pd.DataFrame({'k1': ['one', 'two'] * 
   ....:                      'k2': [1, 1, 2, 3, 3, 4

In [46]: data
Out[46]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
6  two   4
```

DataFrame的duplicated方法返回一个布尔型Series，表示各行是否是重复行（前面出现过的行）：

```
In [47]: data.duplicated()
```

```
Out[47]:
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

还有一个与此相关的drop_duplicates方法，它会返回一个DataFrame，重复的数组会标为False：

```
In [48]: data.drop_duplicates()
Out[48]:
    k1  k2
0  one   1
1  two   1
2  one   2
3  two   3
4  one   3
5  two   4
```

这两个方法默认会判断全部列，你也可以指定部分列进行重复项判断。假设我们还有一列值，且只希望根据k1列过滤重复项：

```
In [49]: data['v1'] = range(7)

In [50]: data.drop_duplicates(['k1'])
Out[50]:
    k1  k2  v1
0  one   1   0
1  two   1   1
```

duplicated和drop_duplicates默认保留的是第一个出现的值组合。传入keep='last'则保留最后一个：

```
In [51]: data.drop_duplicates(['k1', 'k2'], keep='las
Out[51]:
    k1  k2  v1
0  one   1   0
1  two   1   1
2  one   2   2
3  two   3   3
4  one   3   4
6  two   4   6
```

利用函数或映射进行数据转换

对于许多数据集，你可能希望根据数组、Series或DataFrame列中的值来实现转换工作。我们来看看下面这组有关肉类的数据：

```
In [52]: data = pd.DataFrame({'food': ['bacon', 'pull
   ....:                               'Pastrami', 'c
   ....:                               'pastrami', 'h
   ....:                    'ounces': [4, 3, 12, 6, 

In [53]: data
Out[53]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

假设你想要添加一列表示该肉类食物来源的动物类型。我们先编写一个不同肉类到动物的映射：

```
meat_to_animal = {
  'bacon': 'pig',
  'pulled pork': 'pig',
  'pastrami': 'cow',
  'corned beef': 'cow',
  'honey ham': 'pig',
  'nova lox': 'salmon'
}
```

Series的map方法可以接受一个函数或含有映射关系的字典型对象，但是这里有一个小问题，即有些肉类的首字母大写了，而另一些则没有。因此，我们还需要使用Series的str.lower方法，将各个值转换为小写：

```
In [55]: lowercased = data['food'].str.lower()

In [56]: lowercased
Out[56]:
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object

In [57]: data['animal'] = lowercased.map(meat_to_anim

In [58]: data
Out[58]:
          food  ounces  animal
0        bacon     4.0     pig
1  pulled pork     3.0     pig
2        bacon    12.0     pig
```

```
3     Pastrami     6.0     cow
4  corned beef     7.5     cow
5        Bacon     8.0     pig
6     pastrami     3.0     cow
7    honey ham     5.0     pig
8     nova lox     6.0  salmon
```

我们也可以传入一个能够完成全部这些工作的函数：

```
In [59]: data['food'].map(lambda x: meat_to_animal[x..
Out[59]:
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food, dtype: object
```

使用map是一种实现元素级转换以及其他数据清理工作的便捷方式。

替换值

利用fillna方法填充缺失数据可以看做值替换的一种特殊情况。前面已经看到，map可用于修改对象的数据子集，而replace则提供了一种实现该功能的更简单、更灵活的方式。我们来看看下面这个Series：

```
In [60]: data = pd.Series([1., -999., 2., -999., -100

In [61]: data
Out[61]:
```

```
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
```

-999这个值可能是一个表示缺失数据的标记值。要将其替换为pandas能够理解的NA值，我们可以利用replace来产生一个新的Series（除非传入inplace=True）：

```
In [62]: data.replace(-999, np.nan)
Out[62]:
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

如果你希望一次性替换多个值，可以传入一个由待替换值组成的列表以及一个替换值：：

```
In [63]: data.replace([-999, -1000], np.nan)
Out[63]:
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

要让每个值有不同的替换值，可以传递一个替换列表：

```
In [64]: data.replace([-999, -1000], [np.nan, 0])
Out[64]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

传入的参数也可以是字典：

```
In [65]: data.replace({-999: np.nan, -1000: 0})
Out[65]:
0    1.0
1    NaN
2    2.0
3    NaN
4    0.0
5    3.0
dtype: float64
```

> 笔记：data.replace方法与data.str.replace不同，后者做的是字符串的元素级替换。我们会在后面学习Series的字符串方法。

重命名轴索引

跟Series中的值一样，轴标签也可以通过函数或映射进行转换，从而得到一个新的不同标签的对象。轴还可以被就地修改，而无需新建一个数据结构。接下来看看下面这个简单的例子：

```
In [66]: data = pd.DataFrame(np.arange(12).reshape((3
   ....:                          index=['Ohio', 'Colorado
   ....:                          columns=['one', 'two', '
```

跟Series一样，轴索引也有一个map方法：

```
In [67]: transform = lambda x: x[:4].upper()

In [68]: data.index.map(transform)
Out[68]: Index(['OHIO', 'COLO', 'NEW '], dtype='objec
```

你可以将其赋值给index，这样就可以对DataFrame进行就地修改：

```
In [69]: data.index = data.index.map(transform)

In [70]: data
Out[70]:
one  two  three  four
OHIO    0    1      2     3
COLO    4    5      6     7
NEW     8    9     10    11
```

如果想要创建数据集的转换版（而不是修改原始数据），比较实用的方法是rename：

```
In [71]: data.rename(index=str.title, columns=str.upp
Out[71]:
      ONE  TWO  THREE  FOUR
Ohio    0    1      2     3
Colo    4    5      6     7
New     8    9     10    11
```

特别说明一下，rename可以结合字典型对象实现对部分轴标签的更新：

```
In [72]: data.rename(index={'OHIO': 'INDIANA'},
   ....:             columns={'three': 'peekaboo'})
Out[72]:
```

```
one  two  peekaboo  four
INDIANA    0    1         2     3
COLO       4    5         6     7
NEW        8    9        10    11
```

rename可以实现复制DataFrame并对其索引和列标签进行赋值。如果希望就地修改某个数据集，传入inplace=True即可：

```
In [73]: data.rename(index={'OHIO': 'INDIANA'}, inpla

In [74]: data
Out[74]:
         one  two  three  four
INDIANA    0    1      2     3
COLO       4    5      6     7
NEW        8    9     10    11
```

离散化和面元划分

为了便于分析，连续数据常常被离散化或拆分为“面元”（bin）。假设有一组人员数据，而你希望将它们划分为不同的年龄组：

```
In [75]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 
```

接下来将这些数据划分为“18到25”、“26到35”、“35到60”以及“60以上”几个面元。要实现该功能，你需要使用pandas的cut函数：

```
In [76]: bins = [18, 25, 35, 60, 100]

In [77]: cats = pd.cut(ages, bins)

In [78]: cats
Out[78]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35]
```

pandas返回的是一个特殊的Categorical对象。结果展示了pandas.cut划分的面元。你可以将其看做一组表示面元名称的字符串。它的底层含有一个表示不同分类名称的类型数组，以及一个codes属性中的年龄数据的标签：

```
In [79]: cats.codes
Out[79]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], 

In [80]: cats.categories
Out[80]:
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100
              closed='right',
              dtype='interval[int64]')

In [81]: pd.value_counts(cats)
Out[81]:
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

pd.value_counts(cats)是pandas.cut结果的面元计数。

跟"区间"的数学符号一样，圆括号表示开端，而方括号则表示闭端（包括）。哪边是闭端可以通过right=False进行修改：

```
In [82]: pd.cut(ages, [18, 26, 36, 61, 100], right=Fa
Out[82]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..
 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36)
```

你可 以通过传递一个列表或数组到labels，设置自己的面元名称：

```
In [83]: group_names = ['Youth', 'YoungAdult', 'Middl

In [84]: pd.cut(ages, bins, labels=group_names)
Out[84]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAd
dleAged, YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleA
```

如果向cut传入的是面元的数量而不是确切的面元边界，则它会根据数据的最小值和最大值计算等长面元。下面这个例子中，我们将一些均匀分布的数据分成四组：

```
In [85]: data = np.random.rand(20)

In [86]: pd.cut(data, 4, precision=2)
Out[86]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.9
, 0.55], (0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0
Length: 20
Categories (4, interval[float64]): [(0.12, 0.34] < (0
(0.76, 0.97]]
```

选项precision=2，限定小数只有两位。

qcut是一个非常类似于cut的函数，它可以根据样本分位数对数据进行面元划分。根据数据的分布情况，cut可能无法使各个面元中含有相同数量的数据点。而qcut由于使用的是样本分位数，因此可以得到大小基本相等的面元：

```
In [87]: data = np.random.randn(1000)  # Normally dis
```

```
In [88]: cats = pd.qcut(data, 4)  # Cut into quartile

In [89]: cats
Out[89]:
[(-0.0265, 0.62], (0.62, 3.928], (-0.68, -0.0265], (0
, ..., (-0.68, -0.0265], (-0.68, -0.0265], (-2.95, -0
 -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.68] <
 0.62] <
                                    (0.62, 3.928]]

In [90]: pd.value_counts(cats)
Out[90]:
(0.62, 3.928]       250
(-0.0265, 0.62]     250
(-0.68, -0.0265]    250
(-2.95, -0.68]      250
dtype: int64
```

与cut类似，你也可以传递自定义的分位数（0到1之间的数值，包含端点）：

```
In [91]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[91]:
[(-0.0265, 1.286], (-0.0265, 1.286], (-1.187, -0.0265
5, 1.286], ..., (-1.187, -0.0265], (-1.187, -0.0265],
1.286], (-1.187, -0.0265]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -1.187] <
5, 1.286] <
                                    (1.286, 3.928]]
```

本章稍后在讲解聚合和分组运算时会再次用到cut和qcut，因为这两个离散化函数对分位和分组分析非常重要。

检测和过滤异常值

过滤或变换异常值（outlier）在很大程度上就是运用数组运算。来看一个含有正态分布数据的DataFrame：

```
In [92]: data = pd.DataFrame(np.random.randn(1000, 4)

In [93]: data.describe()
Out[93]:
                 0            1            2
count  1000.000000  1000.000000  1000.000000  1000.00(
mean      0.049091     0.026112    -0.002544    -0.05:
std       0.996947     1.007458     0.995232     0.99!
min      -3.645860    -3.184377    -3.745356    -3.42!
25%      -0.599807    -0.612162    -0.687373    -0.74
50%       0.047101    -0.013609    -0.022158    -0.08!
75%       0.756646     0.695298     0.699046     0.62:
max       2.653656     3.525865     2.735527     3.36(
```

假设你想要找出某列中绝对值大小超过3的值：

```
In [94]: col = data[2]

In [95]: col[np.abs(col) > 3]
Out[95]:
41    -3.399312
136   -3.745356
Name: 2, dtype: float64
```

要选出全部含有“超过3或－3的值”的行，你可以在布尔型DataFrame中使用any方法：

```
In [96]: data[(np.abs(data) > 3).any(1)]
Out[96]:
            0         1         2         3
41   0.457246 -0.025907 -3.399312 -0.974657
```

```
60   1.951312  3.260383  0.963301  1.201206
136  0.508391 -0.196713 -3.745356 -1.520113
235 -0.242459 -3.056990  1.918403 -0.578828
258  0.682841  0.326045  0.425384 -3.428254
322  1.179227 -3.184377  1.369891 -1.074833
544 -3.548824  1.553205 -2.186301  1.277104
635 -0.578093  0.193299  1.397822  3.366626
782 -0.207434  3.525865  0.283070  0.544635
803 -3.645860  0.255475 -0.549574 -1.907459
```

根据这些条件，就可以对值进行设置。下面的代码可以将值限制在区间－3到3以内：

```
In [97]: data[np.abs(data) > 3] = np.sign(data) * 3

In [98]: data.describe()
Out[98]:
                 0            1            2            
count  1000.000000  1000.000000  1000.000000  1000.000
mean      0.050286     0.025567    -0.001399    -0.05
std       0.992920     1.004214     0.991414     0.99
min      -3.000000    -3.000000    -3.000000    -3.00
25%      -0.599807    -0.612162    -0.687373    -0.74
50%       0.047101    -0.013609    -0.022158    -0.08
75%       0.756646     0.695298     0.699046     0.62
max       2.653656     3.000000     2.735527     3.00
```

根据数据的值是正还是负，np.sign(data)可以生成1和-1：

```
In [99]: np.sign(data).head()
Out[99]:
     0    1    2    3
0 -1.0  1.0 -1.0  1.0
1  1.0 -1.0  1.0 -1.0
2  1.0  1.0  1.0 -1.0
3 -1.0 -1.0  1.0 -1.0
```

```
4 -1.0  1.0 -1.0 -1.0
```

排列和随机采样

利用numpy.random.permutation函数可以轻松实现对Series或DataFrame的列的排列工作（permuting，随机重排序）。通过需要排列的轴的长度调用permutation，可产生一个表示新顺序的整数数组：

```
In [100]: df = pd.DataFrame(np.arange(5 * 4).reshape(

In [101]: sampler = np.random.permutation(5)

In [102]: sampler
Out[102]: array([3, 1, 4, 2, 0])
```

然后就可以在基于iloc的索引操作或take函数中使用该数组了：

```
In [103]: df
Out[103]:
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19

In [104]: df.take(sampler)
Out[104]:
    0   1   2   3
3  12  13  14  15
1   4   5   6   7
4  16  17  18  19
2   8   9  10  11
0   0   1   2   3
```

如果不想用替换的方式选取随机子集，可以在Series和DataFrame上使用sample方法：

```
In [105]: df.sample(n=3)
Out[105]:
    0   1   2   3
3  12  13  14  15
4  16  17  18  19
2   8   9  10  11
```

要通过替换的方式产生样本（允许重复选择），可以传递replace=True到sample：

```
In [106]: choices = pd.Series([5, 7, -1, 6, 4])

In [107]: draws = choices.sample(n=10, replace=True)

In [108]: draws
Out[108]:
4    4
1    7
4    4
2   -1
0    5
3    6
1    7
4    4
0    5
4    4
dtype: int64
```

计算指标/哑变量

另一种常用于统计建模或机器学习的转换方式是：将分类变量（categorical variable）转换为“哑变量”或“指标矩阵”。

如果DataFrame的某一列中含有k个不同的值，则可以派生出一

个k列矩阵或DataFrame（其值全为1和0）。pandas有一个get_dummies函数可以实现该功能（其实自己动手做一个也不难）。使用之前的一个DataFrame例子：

```
In [109]: df = pd.DataFrame({'key': ['b', 'b', 'a', '
   .....:                    'data1': range(6)})

In [110]: pd.get_dummies(df['key'])
Out[110]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

有时候，你可能想给指标DataFrame的列加上一个前缀，以便能够跟其他数据进行合并。get_dummies的prefix参数可以实现该功能：

```
In [111]: dummies = pd.get_dummies(df['key'], prefix=

In [112]: df_with_dummy = df[['data1']].join(dummies)

In [113]: df_with_dummy
Out[113]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

如果DataFrame中的某行同属于多个分类，则事情就会有点复杂。看一下MovieLens 1M数据集，14章会更深入地研究它：

```
In [114]: mnames = ['movie_id', 'title', 'genres']

In [115]: movies = pd.read_table('datasets/movielens/m
   .....:                         header=None, names=mn

In [116]: movies[:10]
Out[116]:
   movie_id                               title
0         1                    Toy Story (1995)   Anim
1         2                      Jumanji (1995)  Adven
2         3             Grumpier Old Men (1995)
3         4            Waiting to Exhale (1995)
4         5  Father of the Bride Part II (1995)
5         6                         Heat (1995)
6         7                      Sabrina (1995)
7         8                 Tom and Huck (1995)
8         9                 Sudden Death (1995)
Action
9        10                    GoldenEye (1995)     Ac
```

要为每个genre添加指标变量就需要做一些数据规整操作。首先，我们从数据集中抽取出不同的genre值：

```
In [117]: all_genres = []

In [118]: for x in movies.genres:
   .....:     all_genres.extend(x.split('|'))

In [119]: genres = pd.unique(all_genres)
```

现在有：

```
In [120]: genres
```

```
Out[120]:
array(['Animation', "Children's", 'Comedy', 'Adventur
       'Romance', 'Drama', 'Action', 'Crime', 'Thrill
       'Sci-Fi', 'Documentary', 'War', 'Musical', 'My
       'Western'], dtype=object)
```

构建指标DataFrame的方法之一是从一个全零DataFrame开始：

```
In [121]: zero_matrix = np.zeros((len(movies), len(ge

In [122]: dummies = pd.DataFrame(zero_matrix, columns
```

现在，迭代每一部电影，并将dummies各行的条目设为1。要这么做，我们使用dummies.columns来计算每个类型的列索引：

```
In [123]: gen = movies.genres[0]

In [124]: gen.split('|')
Out[124]: ['Animation', "Children's", 'Comedy']

In [125]: dummies.columns.get_indexer(gen.split('|'))
Out[125]: array([0, 1, 2])
```

然后，根据索引，使用.iloc设定值：

```
In [126]: for i, gen in enumerate(movies.genres):
   .....:     indices = dummies.columns.get_indexer(g
   .....:     dummies.iloc[i, indices] = 1
   .....:
```

然后，和以前一样，再将其与movies合并起来：

```
In [127]: movies_windic = movies.join(dummies.add_pre
```

```
In [128]: movies_windic.iloc[0]
Out[128]:
movie_id                                       1
title                           Toy Story (1995)
genres               Animation|Children's|Comedy
Genre_Animation                                1
Genre_Children's                               1
Genre_Comedy                                   1
Genre_Adventure                                0
Genre_Fantasy                                  0
Genre_Romance                                  0
Genre_Drama                                    0
                                ...
Genre_Crime                                    0
Genre_Thriller                                 0
Genre_Horror                                   0
Genre_Sci-Fi                                   0
Genre_Documentary                              0
Genre_War                                      0
Genre_Musical                                  0
Genre_Mystery                                  0
Genre_Film-Noir                                0
Genre_Western                                  0
Name: 0, Length: 21, dtype: object
```

> 笔记：对于很大的数据，用这种方式构建多成员指标变量就会变得非常慢。最好使用更低级的函数，将其写入NumPy数组，然后结果包装在DataFrame中。

一个对统计应用有用的秘诀是：结合get_dummies和诸如cut之类的离散化函数：

```
In [129]: np.random.seed(12345)

In [130]: values = np.random.rand(10)
```

```
In [131]: values
Out[131]:
array([ 0.9296,  0.3164,  0.1839,  0.2046,  0.5677,  (
        0.6532,  0.7489,  0.6536])

In [132]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [133]: pd.get_dummies(pd.cut(values, bins))
Out[133]:
   (0.0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0
0           0           0           0           0
1           0           1           0           0
2           1           0           0           0
3           0           1           0           0
4           0           0           1           0
5           0           0           1           0
6           0           0           0           0
7           0           0           0           1
8           0           0           0           1
9           0           0           0           1
```

我们用numpy.random.seed，使这个例子具有确定性。本书后面会介绍pandas.get_dummies。

7.3 字符串操作

Python能够成为流行的数据处理语言，部分原因是其简单易用的字符串和文本处理功能。大部分文本运算都直接做成了字符串对象的内置方法。对于更为复杂的模式匹配和文本操作，则可能需要用到正则表达式。pandas对此进行了加强，它使你能够对整组数据应用字符串表达式和正则表达式，而且能处理烦人的缺失数据。

字符串对象方法

对于许多字符串处理和脚本应用，内置的字符串方法已经能够满

足要求了。例如，以逗号分隔的字符串可以用split拆分成数段：

```
In [134]: val = 'a,b,  guido'
In [135]: val.split(',')
Out[135]: ['a', 'b', '  guido']
```

split常常与strip一起使用，以去除空白符（包括换行符）：

```
In [136]: pieces = [x.strip() for x in val.split(',')

In [137]: pieces
Out[137]: ['a', 'b', 'guido']
```

利用加法，可以将这些子字符串以双冒号分隔符的形式连接起来：

```
In [138]: first, second, third = pieces

In [139]: first + '::' + second + '::' + third
Out[139]: 'a::b::guido'
```

但这种方式并不是很实用。一种更快更符合Python风格的方式是，向字符串"::"的join方法传入一个列表或元组：

```
In [140]: '::'.join(pieces)
Out[140]: 'a::b::guido'
```

其它方法关注的是子串定位。检测子串的最佳方式是利用Python的in关键字，还可以使用index和find：

```
In [141]: 'guido' in val
Out[141]: True

In [142]: val.index(',')
Out[142]: 1
```

```
In [143]: val.find(':')
Out[143]: -1
```

注意find和index的区别：如果找不到字符串，index将会引发一个异常（而不是返回－1）：

```
In [144]: val.index(':')
---------------------------------------------------------------------
ValueError                                Traceback (m
<ipython-input-144-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

与此相关，count可以返回指定子串的出现次数：

```
In [145]: val.count(',')
Out[145]: 2
```

replace用于将指定模式替换为另一个模式。通过传入空字符串，它也常常用于删除模式：

```
In [146]: val.replace(',', '::')
Out[146]: 'a::b::  guido'

In [147]: val.replace(',', '')
Out[147]: 'ab  guido'
```

表7-3列出了Python内置的字符串方法。

这些运算大部分都能使用正则表达式实现（马上就会看到）。

表7-3：Python内置的字符串方法

方法	说明
count	返回子串在字符串中的出现次数（非重叠）
endswith、startswith	如果字符串以某个后缀结尾（以某个前缀开头），则返回True
join	将字符串用作连接其他字符串序列的分隔符
index	如果在字符串中找到子串，则返回子串第一个字符所在的位置。如果没有找到，则引发ValueError。
find	如果在字符串中找到子串，则返回第一个发现的子串的第一个字符所在的位置。如果没有找到，则返回 −1
rfind	如果在字符串中找到子串，则返回最后一个发现的子串的第一个字符所在的位置。如果没有找到，则返回 −1
replace	用另一个字符串替换指定子串
strip、rstrip、lstrip	去除空白符（包括换行符）。相当于对各个元素执行x.strip()（以及rstrip、lstrip）。译注10
split	通过指定的分隔符将字符串拆分为一组子串
lower、upper	分别将字母字符转换为小写或大写
ljust、rjust	用空格（或其他字符）填充字符串的空白侧以返回符合最低宽度的字符串

casefold 将字符转换为小写，并将任何特定区域的变量字符组合转换成一个通用的可比较形式。

正则表达式

正则表达式提供了一种灵活的在文本中搜索或匹配（通常比前者复杂）字符串模式的方式。正则表达式，常称作regex，是根据正则表达式语言编写的字符串。Python内置的re模块负责对字符串应用正则表达式。我将通过一些例子说明其使用方法。

> 笔记：正则表达式的编写技巧可以自成一章，超出了本书的范围。从网上和其它书可以找到许多非常不错的教程和参考资料。

re模块的函数可以分为三个大类：模式匹配、替换以及拆分。当然，它们之间是相辅相成的。一个regex描述了需要在文本中定

位的一个模式，它可以用于许多目的。我们先来看一个简单的例子：假设我想要拆分一个字符串，分隔符为数量不定的一组空白符（制表符、空格、换行符等）。描述一个或多个空白符的regex是\s+：

```
In [148]: import re

In [149]: text = "foo    bar\t baz  \tqux"

In [150]: re.split('\s+', text)
Out[150]: ['foo', 'bar', 'baz', 'qux']
```

调用re.split('\s+',text)时，正则表达式会先被编译，然后再在text上调用其split方法。你可以用re.compile自己编译regex以得到一个可重用的regex对象：

```
In [151]: regex = re.compile('\s+')

In [152]: regex.split(text)
Out[152]: ['foo', 'bar', 'baz', 'qux']
```

如果只希望得到匹配regex的所有模式，则可以使用findall方法：

```
In [153]: regex.findall(text)
Out[153]: ['    ', '\t ', '  \t']
```

> 笔记：如果想避免正则表达式中不需要的转义（\），则可以使用原始字符串字面量如r'C:\x'（也可以编写其等价式'C:\\x'）。

如果打算对许多字符串应用同一条正则表达式，强烈建议通过re.compile创建regex对象。这样将可以节省大量的CPU时间。

match和search跟findall功能类似。findall返回的是字符串中所有的匹配项，而search则只返回第一个匹配项。match更加严格，它只匹配字符串的首部。来看一个小例子，假设我们有一段文本

以及一条能够识别大部分电子邮件地址的正则表达式：

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# re.IGNORECASE makes the regex case-insensitive
regex = re.compile(pattern, flags=re.IGNORECASE)
```

对text使用findall将得到一组电子邮件地址：

```
In [155]: regex.findall(text)
Out[155]:
['dave@google.com',
 'steve@gmail.com',
 'rob@gmail.com',
 'ryan@yahoo.com']
```

search返回的是文本中第一个电子邮件地址（以特殊的匹配项对象形式返回）。对于上面那个regex，匹配项对象只能告诉我们模式在原字符串中的起始和结束位置：

```
In [156]: m = regex.search(text)

In [157]: m
Out[157]: <_sre.SRE_Match object; span=(5, 20), match

In [158]: text[m.start():m.end()]
Out[158]: 'dave@google.com'
```

regex.match则将返回None，因为它只匹配出现在字符串开头的模式：

```
In [159]: print(regex.match(text))
None
```

相关的，sub方法可以将匹配到的模式替换为指定字符串，并返回所得到的新字符串：

```
In [160]: print(regex.sub('REDACTED', text))
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

假设你不仅想要找出电子邮件地址，还想将各个地址分成3个部分：用户名、域名以及域后缀。要实现此功能，只需将待分段的模式的各部分用圆括号包起来即可：

```
In [161]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\

In [162]: regex = re.compile(pattern, flags=re.IGNORE
```

由这种修改过的正则表达式所产生的匹配项对象，可以通过其groups方法返回一个由模式各段组成的元组：

```
In [163]: m = regex.match('wesm@bright.net')

In [164]: m.groups()
Out[164]: ('wesm', 'bright', 'net')
```

对于带有分组功能的模式，findall会返回一个元组列表：

```
In [165]: regex.findall(text)
Out[165]:
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

sub还能通过诸如\1、\2之类的特殊符号访问各匹配项中的分组。符号\1对应第一个匹配的组，\2对应第二个匹配的组，以此类推：

```
In [166]: print(regex.sub(r'Username: \1, Domain: \2,
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

Python中还有许多的正则表达式，但大部分都超出了本书的范围。表7-4是一个简要概括。

表7-4：正则表达式方法

方法	说明
findall、finditer	返回字符串中所有的非重叠匹配模式。findall返回的是由所有模式组成的列表，而finditer则通过一个迭代器逐个返回
match	从字符串起始位置匹配模式，还可以对模式各部分进行分组。如果匹配到模式，则返回一个匹配项对象，否则返回None
search	扫描整个字符串以匹配模式。如果找到则返回一个匹配项对象。跟match不同，其匹配项可以位于字符串的任意位置，而不仅仅是起始处
split	根据找到的模式将字符串拆分为数段
sub、subn	将字符串中所有的（sub）或前n个（subn）模式替换为指定表达式[译注12]。在替换字符串中可以通过\1、\2等符号表示各分组项

pandas的矢量化字符串函数

清理待分析的散乱数据时，常常需要做一些字符串规整化工作。更为复杂的情况是，含有字符串的列有时还含有缺失数据：

```
In [167]: data = {'Dave': 'dave@google.com', 'Steve':
   .....:         'Rob': 'rob@gmail.com', 'Wes': np.na

In [168]: data = pd.Series(data)
```

```
In [169]: data
Out[169]:
Dave     dave@google.com
Rob        rob@gmail.com
Steve    steve@gmail.com
Wes                  NaN
dtype: object

In [170]: data.isnull()
Out[170]:
Dave     False
Rob      False
Steve    False
Wes       True
dtype: bool
```

通过data.map，所有字符串和正则表达式方法都能被应用于（传入lambda表达式或其他函数）各个值，但是如果存在NA（null）就会报错。为了解决这个问题，Series有一些能够跳过NA值的面向数组方法，进行字符串操作。通过Series的str属性即可访问这些方法。例如，我们可以通过str.contains检查各个电子邮件地址是否含有"gmail"：

```
In [171]: data.str.contains('gmail')
Out[171]:
Dave     False
Rob       True
Steve     True
Wes        NaN
dtype: object
```

也可以使用正则表达式，还可以加上任意re选项（如IGNORECASE）：

```
In [172]: pattern
Out[172]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,
```

```
In [173]: data.str.findall(pattern, flags=re.IGNORECA
Out[173]:
Dave     [(dave, google, com)]
Rob        [(rob, gmail, com)]
Steve    [(steve, gmail, com)]
Wes                        NaN
dtype: object
```

有两个办法可以实现矢量化的元素获取操作：要么使用str.get，要么在str属性上使用索引：

```
In [174]: matches = data.str.match(pattern, flags=re.

In [175]: matches
Out[175]:
Dave     True
Rob      True
Steve    True
Wes       NaN
dtype: object
```

要访问嵌入列表中的元素，我们可以传递索引到这两个函数中：

```
In [176]: matches.str.get(1)
Out[176]:
Dave    NaN
Rob     NaN
Steve   NaN
Wes     NaN
dtype: float64

In [177]: matches.str[0]
Out[177]:
Dave    NaN
Rob     NaN
```

```
Steve   NaN
Wes     NaN
dtype: float64
```

你可以利用这种方法对字符串进行截取：

```
In [178]: data.str[:5]
Out[178]:
Dave     dave@
Rob      rob@g
Steve    steve
Wes        NaN
dtype: object
```

表7-5介绍了更多的pandas字符串方法。

方法	说明
cat	实现元素级的字符串连接操作，可指定分隔符
count	返回表示个字符串是否含有指定模式的布尔型数组
extract	使用带分组的正则表达式从字符串 Series 提取一个或多个字符串，结果是一个 DataFrame，每组有一列
endswith	相当于对每个元素执行 x.endswith(pattern)
startswith	相当于对每个元素执行 x.startswith(pattern)
findall	计算各字符串的模式列表
get	获取各元素的第 i 个字符
isalnum	相当于内置的 str.alnum
isalpha	相当于内置的 str.isalpha
isdecimal	相当于内置的 str.isdecimal
isdigit	相当于内置的 str.isdigit
islower	相当于内置的 str.islower
isnumeric	相当于内置的 str.isnumeric
isupper	相当于内置的 str.isupper
join	根据指定的分隔符将 Series 中各元素的字符串连接起来
len	计算各字符串的长度
lower,upper	转换大小写。相当于对各个元素执行 x.lower()或 x.upper()
match	根据指定的正则表达式对各个元素执行 re.match，返回匹配的组为列表
pad	在字符串的左边、右边或两边添加空白符
center	相当于 pad(side='both')
repeat	重复值。例如，s.str.repeat(3)相当于对各个字符串执行 x*3
replace	用指定字符串替换找到的模式
slice	对 Series 中的各个字符串进行子串截取
split	根据分隔符或正则表达式对字符串进行拆分
strip	去除两边的空白符，包括新行
rstrip	去除右边的空白符
lstrip	去除左边的空白符

表7-5 部分矢量化字符串方法

7.4 总结

高效的数据准备可以让你将更多的时间用于数据分析，花较少的时间用于准备工作，这样就可以极大地提高生产力。我们在本章中学习了许多工具，但覆盖并不全面。下一章，我们会学习pandas的聚合与分组。

第8章 数据规整：聚合、合并和重塑

在许多应用中，数据可能分散在许多文件或数据库中，存储的形式也不利于分析。本章关注可以聚合、合并、重塑数据的方法。

首先，我会介绍pandas的层次化索引，它广泛用于以上操作。然后，我深入介绍了一些特殊的数据操作。在第14章，你可以看到这些工具的多种应用。

8.1 层次化索引

层次化索引（hierarchical indexing）是pandas的一项重要功能，它使你能在一个轴上拥有多个（两个以上）索引级别。抽象点说，它使你能以低维度形式处理高维度数据。我们先来看一个简单的例子：创建一个Series，并用一个由列表或数组组成的列表作为索引：

```
In [9]: data = pd.Series(np.random.randn(9),
   ...:                  index=[['a', 'a', 'a', 'b',
   ...:                         [1, 2, 3, 1, 3, 1, 2,

In [10]: data
Out[10]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

看到的结果是经过美化的带有MultiIndex索引的Series的格式。索引之间的“间隔”表示“直接使用上面的标签”：

```
In [11]: data.index
Out[11]:
MultiIndex(levels=[['a', 'b', 'c', 'd'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3], [0, 1
```

对于一个层次化索引的对象，可以使用所谓的部分索引，使用它选取数据子集的操作更简单：

```
In [12]: data['b']
Out[12]:
1   -0.555730
3    1.965781
dtype: float64

In [13]: data['b':'c']
Out[13]:
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
dtype: float64

In [14]: data.loc[['b', 'd']]
Out[14]:
b  1   -0.555730
   3    1.965781
d  2    0.281746
   3    0.769023
dtype: float64
```

有时甚至还可以在"内层"中进行选取：

```
In [15]: data.loc[:, 2]
Out[15]:
a    0.478943
c    0.092908
```

```
d    0.281746
dtype: float64
```

层次化索引在数据重塑和基于分组的操作（如透视表生成）中扮演着重要的角色。例如，可以通过unstack方法将这段数据重新安排到一个DataFrame中：

```
In [16]: data.unstack()
Out[16]:
          1         2         3
a -0.204708  0.478943 -0.519439
b -0.555730       NaN  1.965781
c  1.393406  0.092908       NaN
d       NaN  0.281746  0.769023
```

unstack的逆运算是stack：

```
In [17]: data.unstack().stack()
Out[17]:
a  1   -0.204708
   2    0.478943
   3   -0.519439
b  1   -0.555730
   3    1.965781
c  1    1.393406
   2    0.092908
d  2    0.281746
   3    0.769023
dtype: float64
```

stack和unstack将在本章后面详细讲解。

对于一个DataFrame，每条轴都可以有分层索引：

```
In [18]: frame = pd.DataFrame(np.arange(12).reshape((4
   ....:                          index=[['a', 'a', 'b',
   ....:                          columns=[['Ohio', 'Ohio
```

```
   ....:                                       ['Green', 'Red

In [19]: frame
Out[19]:
     Ohio     Colorado
    Green Red    Green
a 1     0   1        2
  2     3   4        5
b 1     6   7        8
  2     9  10       11
```

各层都可以有名字（可以是字符串，也可以是别的Python对象）。如果指定了名称，它们就会显示在控制台输出中：

```
In [20]: frame.index.names = ['key1', 'key2']

In [21]: frame.columns.names = ['state', 'color']

In [22]: frame
Out[22]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
     2        3   4        5
b    1        6   7        8
     2        9  10       11
```

> 注意：小心区分索引名state、color与行标签。

有了部分列索引，因此可以轻松选取列分组：

```
In [23]: frame['Ohio']
Out[23]:
color      Green  Red
key1 key2
a    1         0    1
```

```
      2          3    4
b     1          6    7
      2          9   10
```

可以单独创建MultiIndex然后复用。上面那个DataFrame中的（带有分级名称）列可以这样创建：

```
MultiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'],
                       names=['state', 'color'])
```

重排与分级排序

有时，你需要重新调整某条轴上各级别的顺序，或根据指定级别上的值对数据进行排序。swaplevel接受两个级别编号或名称，并返回一个互换了级别的新对象（但数据不会发生变化）：

```
In [24]: frame.swaplevel('key1', 'key2')
Out[24]:
state      Ohio     Colorado
color     Green Red    Green
key2 key1
1    a        0   1        2
2    a        3   4        5
1    b        6   7        8
2    b        9  10       11
```

而sort_index则根据单个级别中的值对数据进行排序。交换级别时，常常也会用到sort_index，这样最终结果就是按照指定顺序进行字母排序了：

```
In [25]: frame.sort_index(level=1)
Out[25]:
state      Ohio     Colorado
color     Green Red    Green
key1 key2
a    1        0   1        2
```

```
b    1         6   7         8
a    2         3   4         5
b    2         9  10        11

In [26]: frame.swaplevel(0, 1).sort_index(level=0)
Out[26]:
state       Ohio     Colorado
color      Green Red    Green
key2 key1
1    a         0   1         2
     b         6   7         8
2    a         3   4         5
     b         9  10        11
```

根据级别汇总统计

许多对DataFrame和Series的描述和汇总统计都有一个level选项，它用于指定在某条轴上求和的级别。再以上面那个DataFrame为例，我们可以根据行或列上的级别来进行求和：

```
In [27]: frame.sum(level='key2')
Out[27]:
state  Ohio     Colorado
color Green Red    Green
key2
1         6   8       10
2        12  14       16

In [28]: frame.sum(level='color', axis=1)
Out[28]:
color       Green  Red
key1 key2
a    1          2    1
     2          8    4
b    1         14    7
     2         20   10
```

这其实是利用了pandas的groupby功能，本书稍后将对其进行详细讲解。

使用DataFrame的列进行索引

人们经常想要将DataFrame的一个或多个列当做行索引来用，或者可能希望将行索引变成DataFrame的列。以下面这个DataFrame为例：

```
In [29]: frame = pd.DataFrame({'a': range(7), 'b': ra
   ....:                       'c': ['one', 'one', 'o
   ....:                             'two', 'two'],
   ....:                       'd': [0, 1, 2, 0, 1, 2

In [30]: frame
Out[30]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

DataFrame的set_index函数会将其一个或多个列转换为行索引，并创建一个新的DataFrame：

```
In [31]: frame2 = frame.set_index(['c', 'd'])

In [32]: frame2
Out[32]:
       a  b
c   d
one 0  0  7
    1  1  6
```

```
    2  2  5
two 0  3  4
    1  4  3
    2  5  2
    3  6  1
```

默认情况下，那些列会从DataFrame中移除，但也可以将其保留下来：

```
In [33]: frame.set_index(['c', 'd'], drop=False)
Out[33]:
       a  b    c  d
c   d
one 0  0  7  one  0
    1  1  6  one  1
    2  2  5  one  2
two 0  3  4  two  0
    1  4  3  two  1
    2  5  2  two  2
    3  6  1  two  3
```

reset_index的功能跟set_index刚好相反，层次化索引的级别会被转移到列里面：

```
In [34]: frame2.reset_index()
Out[34]:
c  d  a  b
0  one  0  0  7
1  one  1  1  6
2  one  2  2  5
3  two  0  3  4
4  two  1  4  3
5  two  2  5  2
6  two  3  6  1
```

8.2 合并数据集

pandas对象中的数据可以通过一些方式进行合并：

- pandas.merge可根据一个或多个键将不同DataFrame中的行连接起来。SQL或其他关系型数据库的用户对此应该会比较熟悉，因为它实现的就是数据库的join操作。
- pandas.concat可以沿着一条轴将多个对象堆叠到一起。
- 实例方法combine_first可以将重复数据编接在一起，用一个对象中的值填充另一个对象中的缺失值。

我将分别对它们进行讲解，并给出一些例子。本书剩余部分的示例中将经常用到它们。

数据库风格的DataFrame合并

数据集的合并（merge）或连接（join）运算是通过一个或多个键将行链接起来的。这些运算是关系型数据库（基于SQL）的核心。pandas的merge函数是对数据应用这些算法的主要切入点。

以一个简单的例子开始：

```
In [35]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', '
   ....:                     'data1': range(7)})

In [36]: df2 = pd.DataFrame({'key': ['a', 'b', 'd'],
   ....:                     'data2': range(3)})

In [37]: df1
Out[37]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b
```

```
In [38]: df2
Out[38]:
   data2 key
0      0   a
1      1   b
2      2   d
```

这是一种多对一的合并。df1中的数据有多个被标记为a和b的行，而df2中key列的每个值则仅对应一行。对这些对象调用merge即可得到：

```
In [39]: pd.merge(df1, df2)
Out[39]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

注意，我并没有指明要用哪个列进行连接。如果没有指定，merge就会将重叠列的列名当做键。不过，最好明确指定一下：

```
In [40]: pd.merge(df1, df2, on='key')
Out[40]:
   data1 key  data2
0      0   b      1
1      1   b      1
2      6   b      1
3      2   a      0
4      4   a      0
5      5   a      0
```

如果两个对象的列名不同，也可以分别进行指定：

```
In [41]: df3 = pd.DataFrame({'lkey': ['b', 'b', 'a',
   ....:                     'data1': range(7)})

In [42]: df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                     'data2': range(3)})

In [43]: pd.merge(df3, df4, left_on='lkey', right_on=
Out[43]:
   data1 lkey  data2 rkey
0      0    b      1    b
1      1    b      1    b
2      6    b      1    b
3      2    a      0    a
4      4    a      0    a
5      5    a      0    a
```

可能你已经注意到了，结果里面c和d以及与之相关的数据消失了。默认情况下，merge做的是“内连接”；结果中的键是交集。其他方式还有"left"、"right"以及"outer"。外连接求取的是键的并集，组合了左连接和右连接的效果：

```
In [44]: pd.merge(df1, df2, how='outer')
Out[44]:
   data1 key  data2
0    0.0   b    1.0
1    1.0   b    1.0
2    6.0   b    1.0
3    2.0   a    0.0
4    4.0   a    0.0
5    5.0   a    0.0
6    3.0   c    NaN
7    NaN   d    2.0
```

表8-1对这些选项进行了总结。

选项	说明
inner	使用两个表都有的键
left	使用左表中所有的键
right	使用右表中所有的键
outer	使用两个表中所有的键

表8-1 不同的连接类型

多对多的合并有些不直观。看下面的例子：

```
In [45]: df1 = pd.DataFrame({'key': ['b', 'b', 'a', '
   ....:                            'data1': range(6)})

In [46]: df2 = pd.DataFrame({'key': ['a', 'b', 'a', '
   ....:                            'data2': range(5)})

In [47]: df1
Out[47]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   b

In [48]: df2
Out[48]:
   data2 key
0      0   a
1      1   b
2      2   a
3      3   b
```

```
4       4   d

In [49]: pd.merge(df1, df2, on='key', how='left')
Out[49]:
    data1 key  data2
0       0   b    1.0
1       0   b    3.0
2       1   b    1.0
3       1   b    3.0
4       2   a    0.0
5       2   a    2.0
6       3   c    NaN
7       4   a    0.0
8       4   a    2.0
9       5   b    1.0
10      5   b    3.0
```

多对多连接产生的是行的笛卡尔积。由于左边的DataFrame有3个"b"行，右边的有2个，所以最终结果中就有6个"b"行。连接方式只影响出现在结果中的不同的键的值：

```
In [50]: pd.merge(df1, df2, how='inner')
Out[50]:
   data1 key  data2
0      0   b      1
1      0   b      3
2      1   b      1
3      1   b      3
4      5   b      1
5      5   b      3
6      2   a      0
7      2   a      2
8      4   a      0
9      4   a      2
```

要根据多个键进行合并，传入一个由列名组成的列表即可：

```
In [51]: left = pd.DataFrame({'key1': ['foo', 'foo',
   ....:                      'key2': ['one', 'two',
   ....:                      'lval': [1, 2, 3]})

In [52]: right = pd.DataFrame({'key1': ['foo', 'foo',
   ....:                       'key2': ['one', 'one',
   ....:                       'rval': [4, 5, 6, 7]})

In [53]: pd.merge(left, right, on=['key1', 'key2'], h
Out[53]:
  key1 key2  lval  rval
0  foo  one   1.0   4.0
1  foo  one   1.0   5.0
2  foo  two   2.0   NaN
3  bar  one   3.0   6.0
4  bar  two   NaN   7.0
```

结果中会出现哪些键组合取决于所选的合并方式，你可以这样来理解：多个键形成一系列元组，并将其当做单个连接键（当然，实际上并不是这么回事）。

> 注意：在进行列－列连接时，DataFrame对象中的索引会被丢弃。

对于合并运算需要考虑的最后一个问题是对重复列名的处理。虽然你可以手工处理列名重叠的问题（查看前面介绍的重命名轴标签），但merge有一个更实用的suffixes选项，用于指定附加到左右两个DataFrame对象的重叠列名上的字符串：

```
In [54]: pd.merge(left, right, on='key1')
Out[54]:
  key1 key2_x  lval key2_y  rval
0  foo    one     1    one     4
1  foo    one     1    one     5
2  foo    two     2    one     4
3  foo    two     2    one     5
```

```
4   bar     one      3    one      6
5   bar     one      3    two      7

In [55]: pd.merge(left, right, on='key1', suffixes=('_
Out[55]:
  key1 key2_left  lval key2_right  rval
0  foo       one     1        one     4
1  foo       one     1        one     5
2  foo       two     2        one     4
3  foo       two     2        one     5
4  bar       one     3        one     6
5  bar       one     3        two     7
```

merge的参数请参见表8-2。使用DataFrame的行索引合并是下一节的主题。

表8-2 merge函数的参数

参数	说明
left	参与合并的左侧DataFrame
right	参与合并的右侧DataFrame
how	“inner”、“outer”、“left”、“right”其中之一。默认为“inner”

参数	说明
on	用于连接的列名。必须存在于左右两个DataFrame对象中。如果未指定，且其他连接键也未指定，则以left和right列名的交集作为连接键
left_on	左侧DataFrame中用作连接键的列
right_on	右侧DataFrame中用作连接键的列
left_index	将左侧的行索引用作其连接键
right_index	类似于left_index
sort	根据连接键对合并后的数据进行排序，默认为True。有时在处理大数据集时，禁用该选项可获得更好的性能
suffixes	字符串值元组，用于追加到重叠列名的末尾，默认为('_x', '_y')。例如，如果左右两个DataFrame对象都有“data”，则结果中就会出现“data_x”和“data_y”
copy	设置为False，可以在某些特殊情况下避免将数据复制到结果数据结构中。默认总是复制

indicator 添加特殊的列_merge，它可以指明每个行的来源，它的值有left_only、right_only或both，根据每行的合并数据的来源。

索引上的合并

有时候，DataFrame中的连接键位于其索引中。在这种情况下，你可以传入left_index=True或right_index=True（或两个都传）以说明索引应该被用作连接键：

```
In [56]: left1 = pd.DataFrame({'key': ['a', 'b', 'a',
   ....:                       'value': range(6)})

In [57]: right1 = pd.DataFrame({'group_val': [3.5, 7]

In [58]: left1
Out[58]:

key  value
0   a      0
1   b      1
2   a      2
```

```
3   a      3
4   b      4
5   c      5

In [59]: right1
Out[59]:
   group_val
a        3.5
b        7.0

In [60]: pd.merge(left1, right1, left_on='key', right_
Out[60]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
```

由于默认的merge方法是求取连接键的交集，因此你可以通过外连接的方式得到它们的并集：

```
In [61]: pd.merge(left1, right1, left_on='key', right_
Out[61]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
5   c      5        NaN
```

对于层次化索引的数据，事情就有点复杂了，因为索引的合并默认是多键合并：

```
In [62]: lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio
```

```
   ....:                                        'Nevada', 'Ne
   ....:                               'key2': [2000, 2001, 2
   ....:                               'data': np.arange(5.)}

In [63]: righth = pd.DataFrame(np.arange(12).reshape(
   ....:                               index=[['Nevada', 'Nev
   ....:                                       'Ohio', 'Ohio'
   ....:                                      [2001, 2000, 20
   ....:                               columns=['event1', 'ev

In [64]: lefth
Out[64]:
   data    key1  key2
0   0.0    Ohio  2000
1   1.0    Ohio  2001
2   2.0    Ohio  2002
3   3.0  Nevada  2001
4   4.0  Nevada  2002

In [65]: righth
Out[65]:
             event1  event2
Nevada 2001       0       1
       2000       2       3
Ohio   2000       4       5
       2000       6       7
       2001       8       9
       2002      10      11
```

这种情况下，你必须以列表的形式指明用作合并键的多个列（注意用how='outer'对重复索引值的处理）：

```
In [66]: pd.merge(lefth, righth, left_on=['key1', 'ke
Out[66]:
   data    key1  key2  event1  event2
0   0.0    Ohio  2000       4       5
```

```
0   0.0    Ohio  2000        6        7
1   1.0    Ohio  2001        8        9
2   2.0    Ohio  2002       10       11
3   3.0  Nevada  2001        0        1

In [67]: pd.merge(lefth, righth, left_on=['key1', 'key
   ....:          right_index=True, how='outer')
Out[67]:
   data    key1  key2  event1  event2
0   0.0    Ohio  2000     4.0     5.0
0   0.0    Ohio  2000     6.0     7.0
1   1.0    Ohio  2001     8.0     9.0
2   2.0    Ohio  2002    10.0    11.0
3   3.0  Nevada  2001     0.0     1.0
4   4.0  Nevada  2002     NaN     NaN
4   NaN  Nevada  2000     2.0     3.0
```

同时使用合并双方的索引也没问题：

```
In [68]: left2 = pd.DataFrame([[1., 2.], [3., 4.], [5
   ....:                      index=['a', 'c', 'e'],
   ....:                      columns=['Ohio', 'Nevad

In [69]: right2 = pd.DataFrame([[7., 8.], [9., 10.],
   ....:                       index=['b', 'c', 'd',
   ....:                       columns=['Missouri', '

In [70]: left2
Out[70]:
   Ohio  Nevada
a   1.0     2.0
c   3.0     4.0
e   5.0     6.0

In [71]: right2
Out[71]:
```

```
   Missouri  Alabama
b       7.0      8.0
c       9.0     10.0
d      11.0     12.0
e      13.0     14.0

In [72]: pd.merge(left2, right2, how='outer', left_in
Out[72]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

DataFrame还有一个便捷的join实例方法，它能更为方便地实现按索引合并。它还可用于合并多个带有相同或相似索引的DataFrame对象，但要求没有重叠的列。在上面那个例子中，我们可以编写：

```
In [73]: left2.join(right2, how='outer')
Out[73]:
   Ohio  Nevada  Missouri  Alabama
a   1.0     2.0       NaN      NaN
b   NaN     NaN       7.0      8.0
c   3.0     4.0       9.0     10.0
d   NaN     NaN      11.0     12.0
e   5.0     6.0      13.0     14.0
```

因为一些历史版本的遗留原因，DataFrame的join方法默认使用的是左连接，保留左边表的行索引。它还支持在调用的DataFrame的列上，连接传递的DataFrame索引：

```
In [74]: left1.join(right1, on='key')
Out[74]:
  key  value  group_val
```

```
0   a      0        3.5
1   b      1        7.0
2   a      2        3.5
3   a      3        3.5
4   b      4        7.0
5   c      5        NaN
```

最后，对于简单的索引合并，你还可以向join传入一组DataFrame，下一节会介绍更为通用的concat函数，也能实现此功能：

```
In [75]: another = pd.DataFrame([[7., 8.], [9., 10.],
   ....:                         index=['a', 'c', 'e',
   ....:                         columns=['New York',
'Oregon'])

In [76]: another
Out[76]:
   New York  Oregon
a       7.0     8.0
c       9.0    10.0
e      11.0    12.0
f      16.0    17.0

In [77]: left2.join([right2, another])
Out[77]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a   1.0     2.0       NaN      NaN       7.0     8.0
c   3.0     4.0       9.0     10.0       9.0    10.0
e   5.0     6.0      13.0     14.0      11.0    12.0

In [78]: left2.join([right2, another], how='outer')
Out[78]:
   Ohio  Nevada  Missouri  Alabama  New York  Oregon
a   1.0     2.0       NaN      NaN       7.0     8.0
b   NaN     NaN       7.0      8.0       NaN     NaN
c   3.0     4.0       9.0     10.0       9.0    10.0
```

```
d    NaN      NaN       11.0      12.0       NaN      NaN
e    5.0      6.0       13.0      14.0      11.0     12.0
f    NaN      NaN        NaN       NaN      16.0     17.0
```

轴向连接

另一种数据合并运算也被称作连接（concatenation）、绑定（binding）或堆叠（stacking）。NumPy的concatenation函数可以用NumPy数组来做：

```
In [79]: arr = np.arange(12).reshape((3, 4))

In [80]: arr
Out[80]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [81]: np.concatenate([arr, arr], axis=1)
Out[81]:
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

对于pandas对象（如Series和DataFrame），带有标签的轴使你能够进一步推广数组的连接运算。具体点说，你还需要考虑以下这些东西：

- 如果对象在其它轴上的索引不同，我们应该合并这些轴的不同元素还是只使用交集？
- 连接的数据集是否需要在结果对象中可识别？
- 连接轴中保存的数据是否需要保留？许多情况下，DataFrame默认的整数标签最好在连接时删掉。

pandas的concat函数提供了一种能够解决这些问题的可靠方式。我将给出一些例子来讲解其使用方式。假设有三个没有重叠索引

的Series：

```
In [82]: s1 = pd.Series([0, 1], index=['a', 'b'])

In [83]: s2 = pd.Series([2, 3, 4], index=['c', 'd', '

In [84]: s3 = pd.Series([5, 6], index=['f', 'g'])
```

对这些对象调用concat可以将值和索引粘合在一起：

```
In [85]: pd.concat([s1, s2, s3])
Out[85]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

默认情况下，concat是在axis=0上工作的，最终产生一个新的Series。如果传入axis=1，则结果就会变成一个DataFrame（axis=1是列）：

```
In [86]: pd.concat([s1, s2, s3], axis=1)
Out[86]:
     0    1    2
a  0.0  NaN  NaN
b  1.0  NaN  NaN
c  NaN  2.0  NaN
d  NaN  3.0  NaN
e  NaN  4.0  NaN
f  NaN  NaN  5.0
g  NaN  NaN  6.0
```

这种情况下，另外的轴上没有重叠，从索引的有序并集（外连接）上就可以看出来。传入join='inner'即可得到它们的交集：

```
In [87]: s4 = pd.concat([s1, s3])

In [88]: s4
Out[88]:
a    0
b    1
f    5
g    6
dtype: int64

In [89]: pd.concat([s1, s4], axis=1)
Out[89]:
     0  1
a  0.0  0
b  1.0  1
f  NaN  5
g  NaN  6

In [90]: pd.concat([s1, s4], axis=1, join='inner')
Out[90]:
   0  1
a  0  0
b  1  1
```

在这个例子中，f和g标签消失了，是因为使用的是join='inner'选项。

你可以通过join_axes指定要在其它轴上使用的索引：

```
In [91]: pd.concat([s1, s4], axis=1, join_axes=[['a',
Out[91]:
     0    1
a  0.0  0.0
c  NaN  NaN
```

```
b  1.0  1.0
e  NaN  NaN
```

不过有个问题，参与连接的片段在结果中区分不开。假设你想要在连接轴上创建一个层次化索引。使用keys参数即可达到这个目的：

```
In [92]: result = pd.concat([s1, s1, s3], keys=['one'

In [93]: result
Out[93]:
one    a    0
       b    1
two    a    0
       b    1
three  f    5
       g    6
dtype: int64

In [94]: result.unstack()
Out[94]:
         a    b    f    g
one    0.0  1.0  NaN  NaN
two    0.0  1.0  NaN  NaN
three  NaN  NaN  5.0  6.0
```

如果沿着axis=1对Series进行合并，则keys就会成为DataFrame的列头：

```
In [95]: pd.concat([s1, s2, s3], axis=1, keys=['one',
Out[95]:
   one  two  three
a  0.0  NaN    NaN
b  1.0  NaN    NaN
c  NaN  2.0    NaN
```

```
d  NaN  3.0    NaN
e  NaN  4.0    NaN
f  NaN  NaN    5.0
g  NaN  NaN    6.0
```

同样的逻辑也适用于DataFrame对象：

```
In [96]: df1 = pd.DataFrame(np.arange(6).reshape(3, 2
   ....:                     columns=['one', 'two'])

In [97]: df2 = pd.DataFrame(5 + np.arange(4).reshape(
   ....:                     columns=['three', 'four']

In [98]: df1
Out[98]:
   one  two
a    0    1
b    2    3
c    4    5

In [99]: df2
Out[99]:
   three  four
a      5     6
c      7     8

In [100]: pd.concat([df1, df2], axis=1, keys=['level1
Out[100]:
  level1     level2
     one two  three four
a      0   1    5.0  6.0
b      2   3    NaN  NaN
c      4   5    7.0  8.0
```

如果传入的不是列表而是一个字典，则字典的键就会被当做keys选项的值：

```
In [101]: pd.concat({'level1': df1, 'level2': df2}, a

Out[101]:
  level1     level2
     one two  three four
a      0   1    5.0  6.0
b      2   3    NaN  NaN
c      4   5    7.0  8.0
```

此外还有两个用于管理层次化索引创建方式的参数（参见表8-3）。举个例子，我们可以用names参数命名创建的轴级别：

```
In [102]: pd.concat([df1, df2], axis=1, keys=['level1
   .....:           names=['upper', 'lower'])
Out[102]:
upper level1     level2
lower    one two  three four
a          0   1    5.0  6.0
b          2   3    NaN  NaN
c          4   5    7.0  8.0
```

最后一个关于DataFrame的问题是，DataFrame的行索引不包含任何相关数据：

```
In [103]: df1 = pd.DataFrame(np.random.randn(3, 4), c

In [104]: df2 = pd.DataFrame(np.random.randn(2, 3), c

In [105]: df1
Out[105]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741
```

```
In [106]: df2
Out[106]:
          b         d         a
0  0.476985  3.248944 -1.021228
1 -0.577087  0.124121  0.302614
```

在这种情况下，传入ignore_index=True即可：

```
In [107]: pd.concat([df1, df2], ignore_index=True)
Out[107]:
          a         b         c         d
0  1.246435  1.007189 -1.296221  0.274992
1  0.228913  1.352917  0.886429 -2.001637
2 -0.371843  1.669025 -0.438570 -0.539741
3 -1.021228  0.476985       NaN  3.248944
4  0.302614 -0.577087       NaN  0.124121
```

参数	说明
objs	参与连接的pandas对象的列表或字典。唯一必需的参数
axis	指明连接的轴向，默认为0
join	“inner”、“outer”其中之一，默认为“outer”。指明其他轴向上的索引是按交集（inner）还是并集（outer）进行合并
join_axes	指明用于其他n-1条轴的索引，不执行并集/交集运算
keys	与连接对象有关的值，用于形成连接轴向上的层次化索引。可以是任意值的列表或数组、元组数组、数组列表（如果将levels设置成多级数组的话）
levels	指定用作层次化索引各级别上的索引，如果设置了keys的话[译注3]
names	用于创建分层级别的名称，如果设置了keys和（或）levels的话
verify_integrity	检查结果对象新轴上的重复情况，如果发现则引发异常。默认（False）允许重复
ignore_index	不保留连接轴上的索引，产生一组新索引range(total_length)

表8-3 concat函数的参数

合并重叠数据

还有一种数据组合问题不能用简单的合并（merge）或连接

（concatenation）运算来处理。比如说，你可能有索引全部或部分重叠的两个数据集。举个有启发性的例子，我们使用NumPy的where函数，它表示一种等价于面向数组的if-else：

```
In [108]: a = pd.Series([np.nan, 2.5, np.nan, 3.5, 4.
   .....:               index=['f', 'e', 'd', 'c', 'b

In [109]: b = pd.Series(np.arange(len(a), dtype=np.fl
   .....:               index=['f', 'e', 'd', 'c', 'b

In [110]: b[-1] = np.nan

In [111]: a
Out[111]:
f    NaN
e    2.5
d    NaN
c    3.5
b    4.5
a    NaN
dtype: float64

In [112]: b
Out[112]:
f    0.0
e    1.0
d    2.0
c    3.0
b    4.0
a    NaN
dtype: float64

In [113]: np.where(pd.isnull(a), b, a)
Out[113]: array([ 0. ,  2.5,  2. ,  3.5,  4.5,  nan])
```

Series有一个combine_first方法，实现的也是一样的功能，还带

有pandas的数据对齐：

```
In [114]: b[:-2].combine_first(a[2:])
Out[114]:
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
dtype: float64
```

对于DataFrame，combine_first自然也会在列上做同样的事情，因此你可以将其看做：用传递对象中的数据为调用对象的缺失数据“打补丁”：

```
In [115]: df1 = pd.DataFrame({'a': [1., np.nan, 5., np
   .....:                     'b': [np.nan, 2., np.nan
   .....:                     'c': range(2, 18, 4)})

In [116]: df2 = pd.DataFrame({'a': [5., 4., np.nan, 3
   .....:                     'b': [np.nan, 3., 4., 6

In [117]: df1
Out[117]:
     a    b   c
0  1.0  NaN   2
1  NaN  2.0   6
2  5.0  NaN  10
3  NaN  6.0  14

In [118]: df2
Out[118]:
     a    b
0  5.0  NaN
1  4.0  3.0
2  NaN  4.0
```

```
3  3.0  6.0
4  7.0  8.0

In [119]: df1.combine_first(df2)
Out[119]:
     a    b     c
0  1.0  NaN   2.0
1  4.0  2.0   6.0
2  5.0  4.0  10.0
3  3.0  6.0  14.0
4  7.0  8.0   NaN
```

8.3 重塑和轴向旋转

有许多用于重新排列表格型数据的基础运算。这些函数也称作重塑（reshape）或轴向旋转（pivot）运算。

重塑层次化索引

层次化索引为DataFrame数据的重排任务提供了一种具有良好一致性的方式。主要功能有二：

- stack：将数据的列“旋转”为行。
- unstack：将数据的行“旋转”为列。

我将通过一系列的范例来讲解这些操作。接下来看一个简单的DataFrame，其中的行列索引均为字符串数组：

```
In [120]: data = pd.DataFrame(np.arange(6).reshape((2
   .....:                     index=pd.Index(['Ohio',
   .....:                     columns=pd.Index(['one'
   .....:                     name='number'))

In [121]: data
Out[121]:
number    one  two  three
state
```

```
Ohio        0    1      2
Colorado    3    4      5
```

对该数据使用stack方法即可将列转换为行，得到一个Series：

```
In [122]: result = data.stack()

In [123]: result
Out[123]:
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int64
```

对于一个层次化索引的Series，你可以用unstack将其重排为一个DataFrame：

```
In [124]: result.unstack()
Out[124]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

默认情况下，unstack操作的是最内层（stack也是如此）。传入分层级别的编号或名称即可对其它级别进行unstack操作：

```
In [125]: result.unstack(0)
Out[125]:
state   Ohio  Colorado
number
one        0         3
```

```
two         1          4
three       2          5

In [126]: result.unstack('state')
Out[126]:
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5
```

如果不是所有的级别值都能在各分组中找到的话，则unstack操作可能会引入缺失数据：

```
In [127]: s1 = pd.Series([0, 1, 2, 3], index=['a', 'b

In [128]: s2 = pd.Series([4, 5, 6], index=['c', 'd',

In [129]: data2 = pd.concat([s1, s2], keys=['one', 't

In [130]: data2
Out[130]:
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64

In [131]: data2.unstack()
Out[131]:
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0
```

stack默认会滤除缺失数据，因此该运算是可逆的：

```
In [132]: data2.unstack()
Out[132]:
       a    b    c    d    e
one  0.0  1.0  2.0  3.0  NaN
two  NaN  NaN  4.0  5.0  6.0

In [133]: data2.unstack().stack()
Out[133]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
two  c    4.0
     d    5.0
     e    6.0
dtype: float64

In [134]: data2.unstack().stack(dropna=False)
Out[134]:
one  a    0.0
     b    1.0
     c    2.0
     d    3.0
     e    NaN
two  a    NaN
     b    NaN
     c    4.0
     d    5.0
     e    6.0
dtype: float64
```

在对DataFrame进行unstack操作时，作为旋转轴的级别将会成为结果中的最低级别：

```
In [135]: df = pd.DataFrame({'left': result, 'right':
```

```
   .....:                         columns=pd.Index(['left',

In [136]: df
Out[136]:
side             left  right
state    number
Ohio     one        0      5
         two        1      6
         three      2      7
Colorado one        3      8
         two        4      9
         three      5     10

In [137]: df.unstack('state')
Out[137]:
side   left           right
state  Ohio Colorado  Ohio Colorado
number
one       0        3     5        8
two       1        4     6        9
three     2        5     7       10
```

当调用stack，我们可以指明轴的名字：

```
In [138]: df.unstack('state').stack('side')
Out[138]:
state         Colorado  Ohio
number side
one    left          3     0
       right         8     5
two    left          4     1
       right         9     6
three  left          5     2
       right        10     7
```

将“长格式”旋转为“宽格式”

多个时间序列数据通常是以所谓的“长格式”（long）或“堆叠格式”（stacked）存储在数据库和CSV中的。我们先加载一些示例数据，做一些时间序列规整和数据清洗：

```
In [139]: data = pd.read_csv('examples/macrodata.csv'

In [140]: data.head()
Out[140]:
     year  quarter   realgdp  realcons  realinv  reale
0  1959.0      1.0  2710.349    1707.4  286.898   470
1  1959.0      2.0  2778.801    1733.7  310.859   481
2  1959.0      3.0  2775.488    1751.8  289.226   491
3  1959.0      4.0  2785.204    1753.7  299.356   484
4  1960.0      1.0  2847.699    1770.5  331.722   462
      m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82    5.8  177.146  0.00     0.00
1  141.7      3.08    5.1  177.830  2.34     0.74
2  140.5      3.82    5.3  178.657  2.74     1.09
3  140.0      4.33    5.6  179.386  0.27     4.06
4  139.6      3.50    5.2  180.007  2.31     1.19

In [141]: periods = pd.PeriodIndex(year=data.year, qu
   .....:                          name='date')

In [142]: columns = pd.Index(['realgdp', 'infl', 'une

In [143]: data = data.reindex(columns=columns)

In [144]: data.index = periods.to_timestamp('D', 'end

In [145]: ldata = data.stack().reset_index().rename(c
```

这就是多个时间序列（或者其它带有两个或多个键的可观察数据，这里，我们的键是date和item）的长格式。表中的每行代表一次观察。

关系型数据库（如MySQL）中的数据经常都是这样存储的，因为固定架构（即列名和数据类型）有一个好处：随着表中数据的添加，item列中的值的种类能够增加。在前面的例子中，date和item通常就是主键（用关系型数据库的说法），不仅提供了关系完整性，而且提供了更为简单的查询支持。有的情况下，使用这样的数据会很麻烦，你可能会更喜欢DataFrame，不同的item值分别形成一列，date列中的时间戳则用作索引。DataFrame的pivot方法完全可以实现这个转换：

```
In [147]: pivoted = ldata.pivot('date', 'item', 'valu

In [148]: pivoted
Out[148]:
item         infl    realgdp  unemp
date
1959-03-31  0.00   2710.349    5.8
1959-06-30  2.34   2778.801    5.1
1959-09-30  2.74   2775.488    5.3
1959-12-31  0.27   2785.204    5.6
1960-03-31  2.31   2847.699    5.2
1960-06-30  0.14   2834.390    5.2
1960-09-30  2.70   2839.022    5.6
1960-12-31  1.21   2802.616    6.3
1961-03-31 -0.40   2819.264    6.8
1961-06-30  1.47   2872.005    7.0
...          ...        ...    ...
2007-06-30  2.75  13203.977    4.5
2007-09-30  3.45  13321.109    4.7
2007-12-31  6.38  13391.249    4.8
2008-03-31  2.82  13366.865    4.9
2008-06-30  8.53  13415.266    5.4
2008-09-30 -3.16  13324.600    6.0
2008-12-31 -8.79  13141.920    6.9
2009-03-31  0.94  12925.410    8.1
2009-06-30  3.37  12901.504    9.2
2009-09-30  3.56  12990.341    9.6
```

```
[203 rows x 3 columns]
```

前两个传递的值分别用作行和列索引，最后一个可选值则是用于填充DataFrame的数据列。假设有两个需要同时重塑的数据列：

```
In [149]: ldata['value2'] = np.random.randn(len(ldata

In [150]: ldata[:10]
Out[150]:
        date     item     value    value2
0 1959-03-31  realgdp  2710.349  0.523772
1 1959-03-31     infl     0.000  0.000940
2 1959-03-31    unemp     5.800  1.343810
3 1959-06-30  realgdp  2778.801 -0.713544
4 1959-06-30     infl     2.340 -0.831154
5 1959-06-30    unemp     5.100 -2.370232
6 1959-09-30  realgdp  2775.488 -1.860761
7 1959-09-30     infl     2.740 -0.860757
8 1959-09-30    unemp     5.300  0.560145
9 1959-12-31  realgdp  2785.204 -1.265934
```

如果忽略最后一个参数，得到的DataFrame就会带有层次化的列：

```
In [151]: pivoted = ldata.pivot('date', 'item')

In [152]: pivoted[:5]
Out[152]:
           value                        value2
item        infl   realgdp unemp      infl   realgdp
date
1959-03-31  0.00  2710.349   5.8  0.000940  0.523772
1959-06-30  2.34  2778.801   5.1 -0.831154 -0.713544
1959-09-30  2.74  2775.488   5.3 -0.860757 -1.860761
1959-12-31  0.27  2785.204   5.6  0.119827 -1.265934
1960-03-31  2.31  2847.699   5.2 -2.359419  0.332883
```

```
In [153]: pivoted['value'][:5]
Out[153]:
item         infl   realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

注意，pivot其实就是用set_index创建层次化索引，再用unstack重塑：

```
In [154]: unstacked = ldata.set_index(['date', 'item'

In [155]: unstacked[:7]
Out[155]:
           value                         value2
item        infl   realgdp unemp       infl   realgdp
date
1959-03-31  0.00  2710.349   5.8  0.000940  0.523772
1959-06-30  2.34  2778.801   5.1 -0.831154 -0.713544
1959-09-30  2.74  2775.488   5.3 -0.860757 -1.860761
1959-12-31  0.27  2785.204   5.6  0.119827 -1.265934
1960-03-31  2.31  2847.699   5.2 -2.359419  0.332883
1960-06-30  0.14  2834.390   5.2 -0.970736 -1.541996
1960-09-30  2.70  2839.022   5.6  0.377984  0.286350
```

将“宽格式”旋转为“长格式”

旋转DataFrame的逆运算是pandas.melt。它不是将一列转换到多个新的DataFrame，而是合并多个列成为一个，产生一个比输入长的DataFrame。看一个例子：

```
In [157]: df = pd.DataFrame({'key': ['foo', 'bar', 'ba
```

```
   .....:                               'A': [1, 2, 3],
   .....:                               'B': [4, 5, 6],
   .....:                               'C': [7, 8, 9]})

In [158]: df
Out[158]:
   A  B  C  key
0  1  4  7  foo
1  2  5  8  bar
2  3  6  9  baz
```

key列可能是分组指标，其它的列是数据值。当使用pandas.melt，我们必须指明哪些列是分组指标。下面使用key作为唯一的分组指标：

```
In [159]: melted = pd.melt(df, ['key'])

In [160]: melted
Out[160]:
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
6  foo        C      7
7  bar        C      8
8  baz        C      9
```

使用pivot，可以重塑回原来的样子：

```
In [161]: reshaped = melted.pivot('key', 'variable',

In [162]: reshaped
Out[162]:
```

```
variable  A  B  C
key
bar       2  5  8
baz       3  6  9
foo       1  4  7
```

因为pivot的结果从列创建了一个索引，用作行标签，我们可以使用reset_index将数据移回列：

```
In [163]: reshaped.reset_index()
Out[163]:
variable  key  A  B  C
0         bar  2  5  8
1         baz  3  6  9
2         foo  1  4  7
```

你还可以指定列的子集，作为值的列：

```
In [164]: pd.melt(df, id_vars=['key'], value_vars=['A
Out[164]:
  key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
```

pandas.melt也可以不用分组指标：

```
In [165]: pd.melt(df, value_vars=['A', 'B', 'C'])
Out[165]:
  variable  value
0        A      1
1        A      2
2        A      3
```

```
3         B      4
4         B      5
5         B      6
6         C      7
7         C      8
8         C      9

In [166]: pd.melt(df, value_vars=['key', 'A', 'B'])
Out[166]:
  variable value
0      key   foo
1      key   bar
2      key   baz
3        A     1
4        A     2
5        A     3
6        B     4
7        B     5
8        B     6
```

8.4 总结

现在你已经掌握了pandas数据导入、清洗、重塑，我们可以进一步学习matplotlib数据可视化。我们在稍后会回到pandas，学习更高级的分析。

第9章 绘图和可视化

信息可视化（也叫绘图）是数据分析中最重要的工作之一。它可能是探索过程的一部分，例如，帮助我们找出异常值、必要的数据转换、得出有关模型的idea等。另外，做一个可交互的数据可视化也许是工作的最终目标。Python有许多库进行静态或动态的数据可视化，但我这里重要关注于matplotlib（http://matplotlib.org/）和基于它的库。

matplotlib是一个用于创建出版质量图表的桌面绘图包（主要是2D方面）。该项目是由John Hunter于2002年启动的，其目的是为Python构建一个MATLAB式的绘图接口。matplotlib和IPython社区进行合作，简化了从IPython shell（包括现在的Jupyter notebook）进行交互式绘图。matplotlib支持各种操作系统上许多不同的GUI后端，而且还能将图片导出为各种常见的矢量（vector）和光栅（raster）图：PDF、SVG、JPG、PNG、BMP、GIF等。除了几张，本书中的大部分图都是用它生成的。

随着时间的发展，matplotlib衍生出了多个数据可视化的工具集，它们使用matplotlib作为底层。其中之一是seaborn（http://seaborn.pydata.org/），本章后面会学习它。

学习本章代码案例的最简单方法是在Jupyter notebook进行交互式绘图。在Jupyter notebook中执行下面的语句：

```
%matplotlib notebook
```

9.1 matplotlib API入门

matplotlib的通常引入约定是：

```
In [11]: import matplotlib.pyplot as plt
```

在Jupyter中运行%matplotlib notebook（或在IPython中运行%matplotlib），就可以创建一个简单的图形。如果一切设置正确，会看到图9-1：

```
In [12]: import numpy as np

In [13]: data = np.arange(10)

In [14]: data
Out[14]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [15]: plt.plot(data)
```

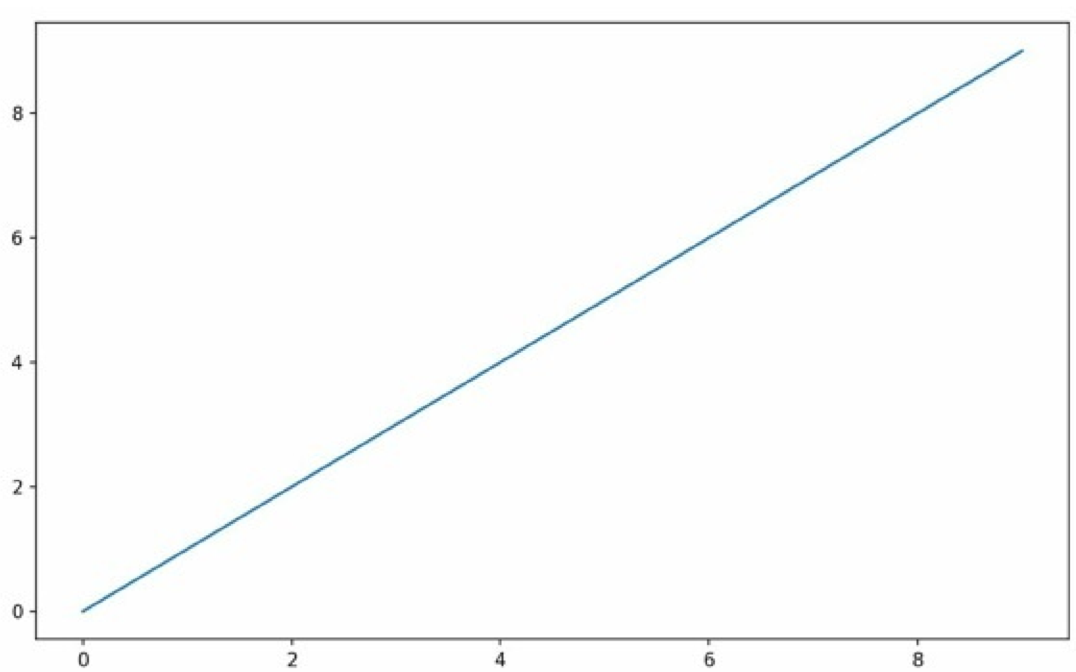


图9-1 简单的线图

虽然seaborn这样的库和pandas的内置绘图函数能够处理许多普通的绘图任务，但如果需要自定义一些高级功能的话就必须学习matplotlib API。

> 笔记：虽然本书没有详细地讨论matplotlib的各种功能，但足以将你引入门。matplotlib的示例库和文档是学习高级特性的最好资源。

Figure和Subplot

matplotlib的图像都位于Figure对象中。你可以用plt.figure创建一

个新的Figure：

```
In [16]: fig = plt.figure()
```

如果用的是IPython，这时会弹出一个空窗口，但在Jupyter中，必须再输入更多命令才能看到。plt.figure有一些选项，特别是figsize，它用于确保当图片保存到磁盘时具有一定的大小和纵横比。

不能通过空Figure绘图。必须用add_subplot创建一个或多个subplot才行：

```
In [17]: ax1 = fig.add_subplot(2, 2, 1)
```

这条代码的意思是：图像应该是2×2的（即最多4张图），且当前选中的是4个subplot中的第一个（编号从1开始）。如果再把后面两个subplot也创建出来，最终得到的图像如图9-2所示：

```
In [18]: ax2 = fig.add_subplot(2, 2, 2)

In [19]: ax3 = fig.add_subplot(2, 2, 3)
```

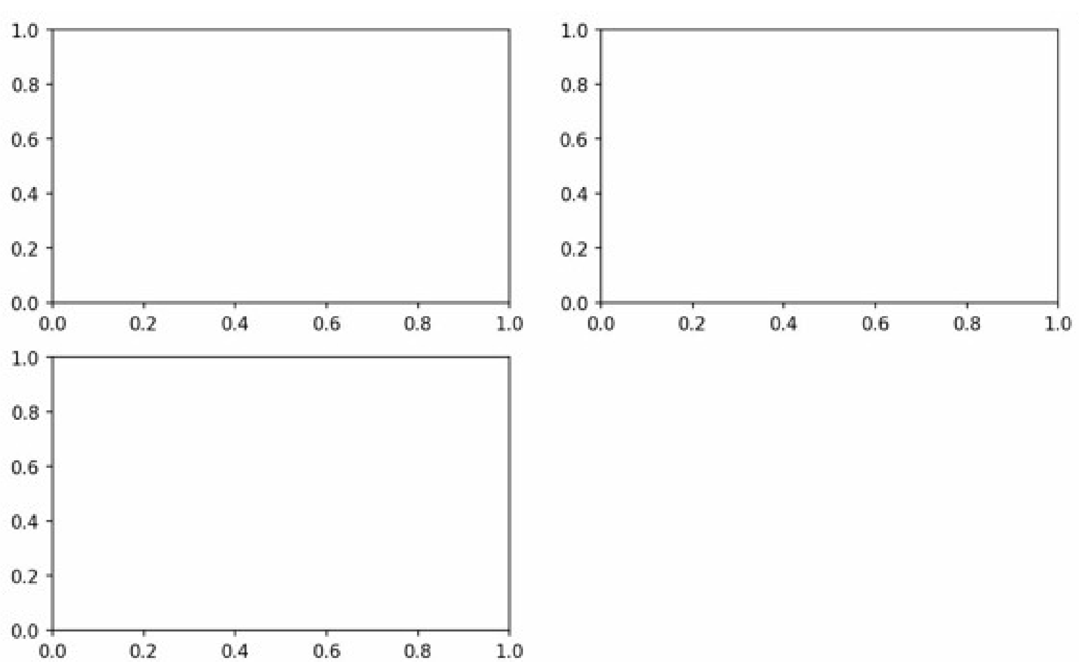


图9-2 带有三个subplot的Figure

> 提示：使用Jupyter notebook有一点不同，即每个小窗重新执行后，图形会被重置。因此，对于复杂的图形，，你必须将所有的绘图命令存在一个小窗里。

这里，我们运行同一个小窗里的所有命令：

```
fig = plt.figure()
ax1 = fig.add_subplot(2, 2, 1)
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
```

如果这时执行一条绘图命令（如plt.plot([1.5, 3.5, -2, 1.6])），matplotlib就会在最后一个用过的subplot（如果没有则创建一个）上进行绘制，隐藏创建figure和subplot的过程。因此，如果我们执行下列命令，你就会得到如图9-3所示的结果：

```
In [20]: plt.plot(np.random.randn(50).cumsum(), 'k--'
```

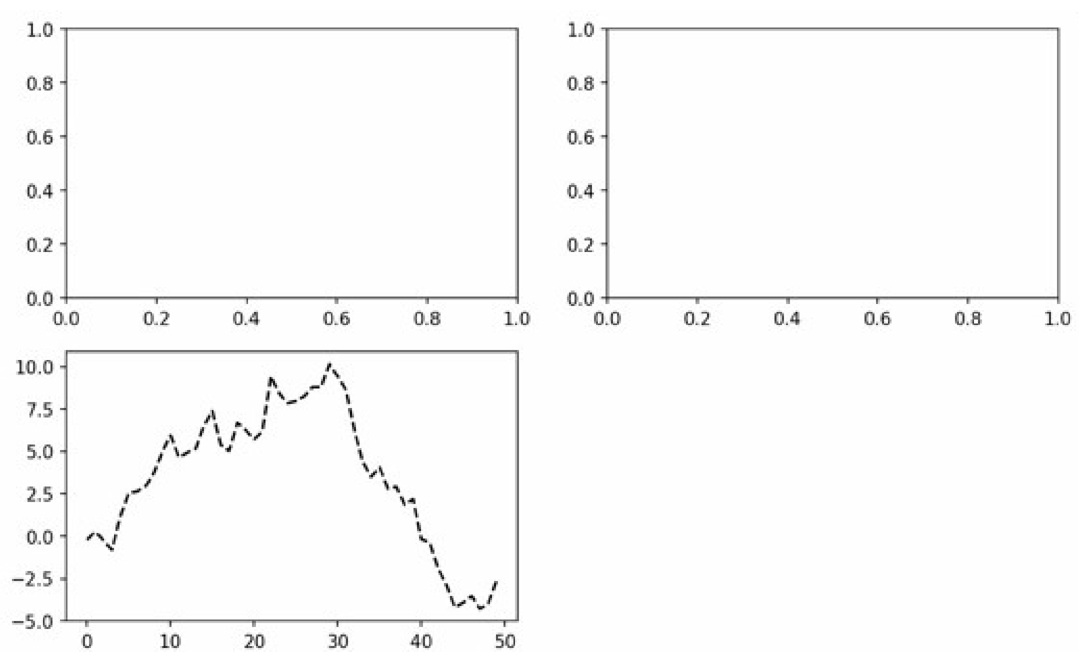


图9-3 绘制一次之后的图像

"k--"是一个线型选项，用于告诉matplotlib绘制黑色虚线图。上面那些由fig.add_subplot所返回的对象是AxesSubplot对象，直接调用它们的实例方法就可以在其它空着的格子里面画图了，如图9-4所示：

```
In [21]: _ = ax1.hist(np.random.randn(100), bins=20,

In [22]: ax2.scatter(np.arange(30), np.arange(30) + 3
```

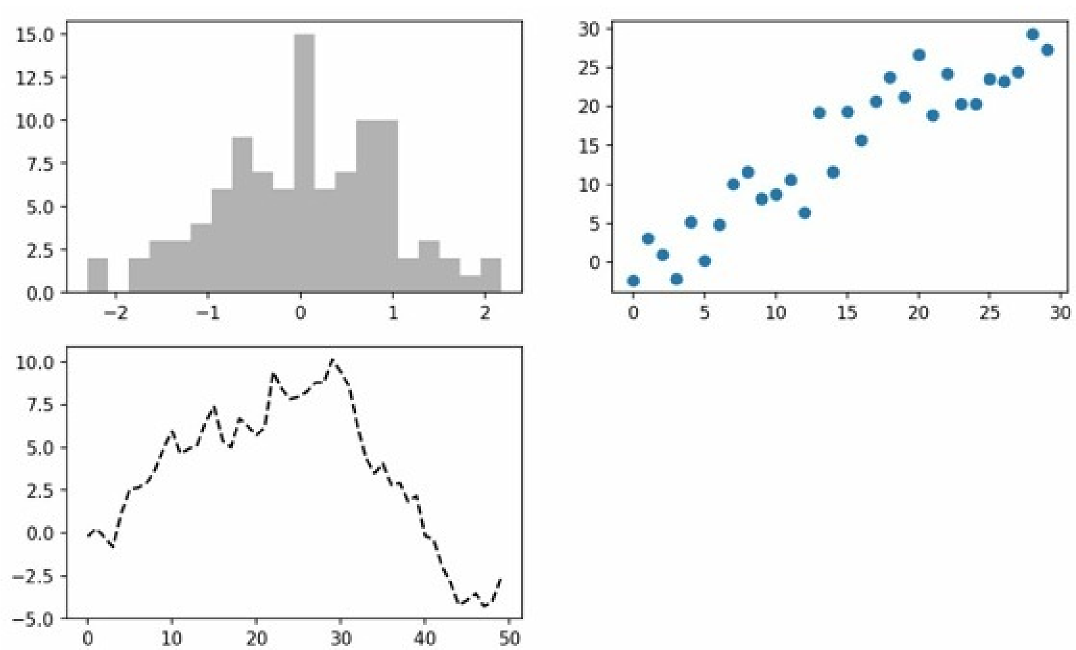


图9-4 继续绘制两次之后的图像

你可以在matplotlib的文档中找到各种图表类型。

创建包含subplot网格的figure是一个非常常见的任务，matplotlib有一个更为方便的方法plt.subplots，它可以创建一个新的Figure，并返回一个含有已创建的subplot对象的NumPy数组：

```
In [24]: fig, axes = plt.subplots(2, 3)

In [25]: axes
Out[25]:
array([[<matplotlib.axes._subplots.AxesSubplot object
```

```
        <matplotlib.axes._subplots.AxesSubplot object
        <matplotlib.axes._subplots.AxesSubplot object
       [<matplotlib.axes._subplots.AxesSubplot object
        <matplotlib.axes._subplots.AxesSubplot object
        <matplotlib.axes._subplots.AxesSubplot object
=object)
```

这是非常实用的，因为可以轻松地对axes数组进行索引，就好像是一个二维数组一样，例如axes[0,1]。你还可以通过sharex和sharey指定subplot应该具有相同的X轴或Y轴。在比较相同范围的数据时，这也是非常实用的，否则，matplotlib会自动缩放各图表的界限。有关该方法的更多信息，请参见表9-1。

参数	说明
nrows	subplot的行数
ncols	subplot的列数
sharex	所有subplot应该使用相同的X轴刻度（调节xlim将会影响所有subplot）
sharey	所有subplot应该使用相同的Y轴刻度（调节ylim将会影响所有subplot）
subplot_kw	用于创建各subplot的关键字字典
**fig_kw	创建figure时的其他关键字，如plt.subplots(2,2,figsize=(8,6))

表9-1 pyplot.subplots的选项

调整subplot周围的间距

默认情况下，matplotlib会在subplot外围留下一定的边距，并在subplot之间留下一定的间距。间距跟图像的高度和宽度有关，因此，如果你调整了图像大小（不管是编程还是手工），间距也会自动调整。利用Figure的subplots_adjust方法可以轻而易举地修改间距，此外，它也是个顶级函数：

```
subplots_adjust(left=None, bottom=None, right=None, t
                wspace=None, hspace=None)
```

wspace和hspace用于控制宽度和高度的百分比，可以用作subplot之间的间距。下面是一个简单的例子，其中我将间距收缩

到了0（如图9-5所示）：

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=Tr
for i in range(2):
    for j in range(2):
        axes[i, j].hist(np.random.randn(500), bins=50
plt.subplots_adjust(wspace=0, hspace=0)
```

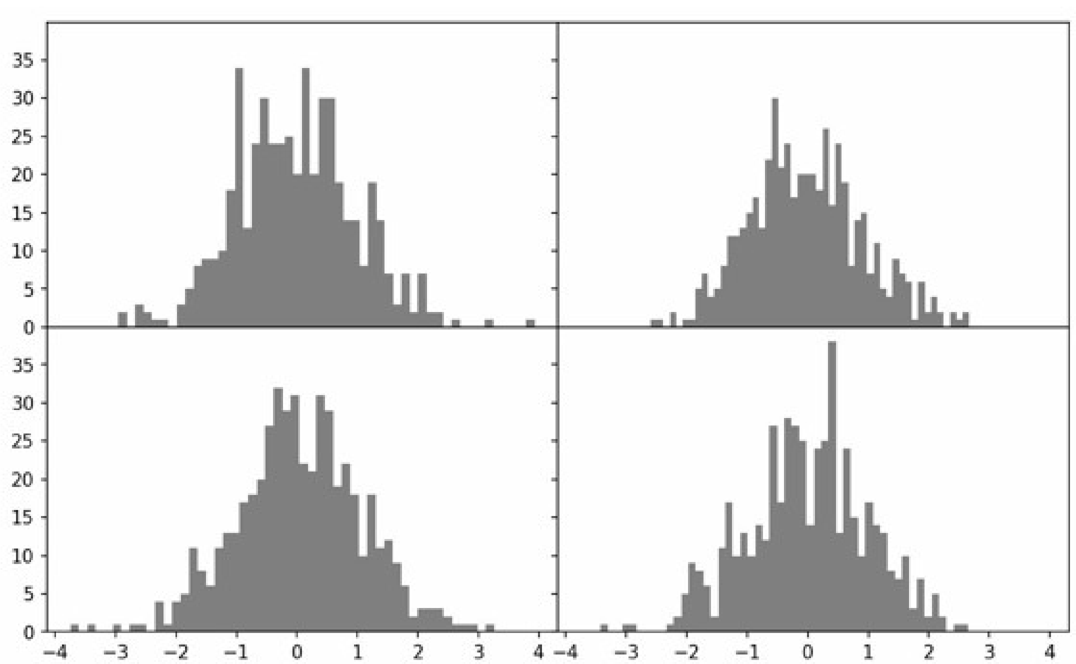


图9-5 各subplot之间没有间距

不难看出，其中的轴标签重叠了。matplotlib不会检查标签是否重叠，所以对于这种情况，你只能自己设定刻度位置和刻度标签。后面几节将会详细介绍该内容。

颜色、标记和线型

matplotlib的plot函数接受一组X和Y坐标，还可以接受一个表示颜色和线型的字符串缩写。例如，要根据x和y绘制绿色虚线，你可以执行如下代码：

```
ax.plot(x, y, 'g--')
```

这种在一个字符串中指定颜色和线型的方式非常方便。在实际中，如果你是用代码绘图，你可能不想通过处理字符串来获得想要的格式。通过下面这种更为明确的方式也能得到同样的效果：

```
ax.plot(x, y, linestyle='--', color='g')
```

常用的颜色可以使用颜色缩写，你也可以指定颜色码（例如，'#CECECE'）。你可以通过查看plot的文档字符串查看所有线型的合集（在IPython和Jupyter中使用plot?）。

线图可以使用标记强调数据点。因为matplotlib可以创建连续线图，在点之间进行插值，因此有时可能不太容易看出真实数据点的位置。标记也可以放到格式字符串中，但标记类型和线型必须放在颜色后面（见图9-6）：

```
In [30]: from numpy.random import randn

In [31]: plt.plot(randn(30).cumsum(), 'ko--')
```

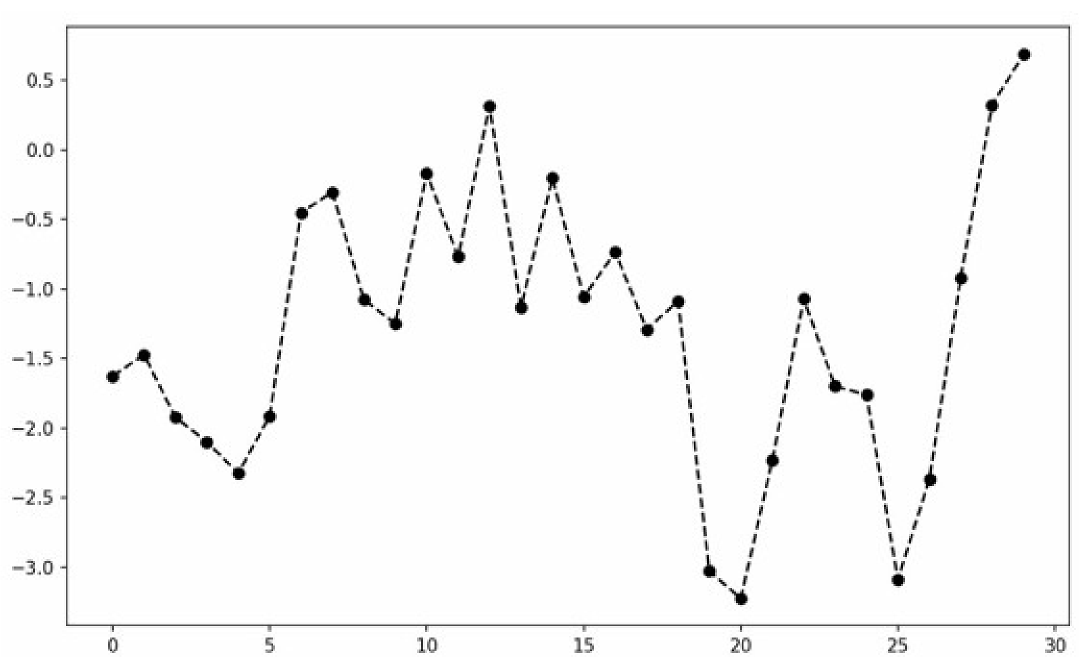


图9-6 带有标记的线型图示例

还可以将其写成更为明确的形式：

```
plot(randn(30).cumsum(), color='k', linestyle='dashed
```

在线型图中，非实际数据点默认是按线性方式插值的。可以通过drawstyle选项修改（见图9-7）：

```
In [33]: data = np.random.randn(30).cumsum()

In [34]: plt.plot(data, 'k--', label='Default')
Out[34]: [<matplotlib.lines.Line2D at 0x7fb624d86160>

In [35]: plt.plot(data, 'k-', drawstyle='steps-post',
Out[35]: [<matplotlib.lines.Line2D at 0x7fb624d869e8>

In [36]: plt.legend(loc='best')
```

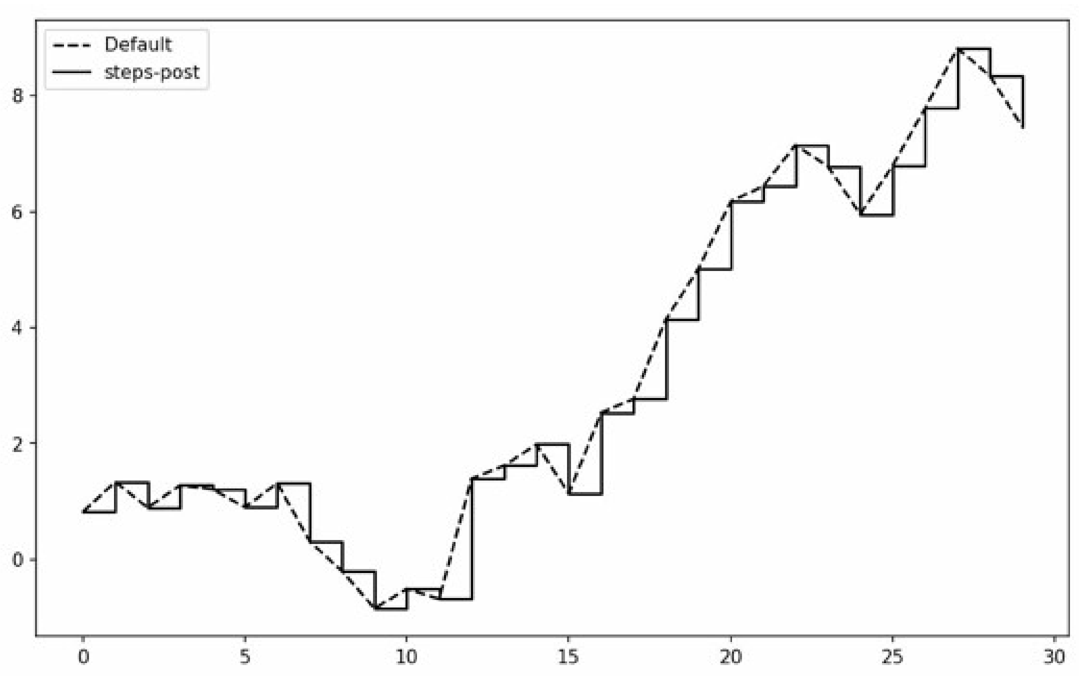


图9-7 不同drawstyle选项的线型图

你可能注意到运行上面代码时有输出<matplotlib.lines.Line2D at ...>。matplotlib会返回引用了新添加的子组件的对象。大多数时候，你可以放心地忽略这些输出。这里，因为我们传递了label参数到plot，我们可以创建一个plot图例，指明每条使用plt.legend

的线。

> 笔记：你必须调用plt.legend（或使用ax.legend，如果引用了轴的话）来创建图例，无论你绘图时是否传递label标签选项。

刻度、标签和图例

对于大多数的图表装饰项，其主要实现方式有二：使用过程型的pyplot接口（例如，matplotlib.pyplot）以及更为面向对象的原生matplotlib API。

pyplot接口的设计目的就是交互式使用，含有诸如xlim、xticks和xticklabels之类的方法。它们分别控制图表的范围、刻度位置、刻度标签等。其使用方式有以下两种：

- 调用时不带参数，则返回当前的参数值（例如，plt.xlim()返回当前的X轴绘图范围）。
- 调用时带参数，则设置参数值（例如，plt.xlim([0,10])会将X轴的范围设置为0到10）。

所有这些方法都是对当前或最近创建的AxesSubplot起作用的。它们各自对应subplot对象上的两个方法，以xlim为例，就是ax.get_xlim和ax.set_xlim。我更喜欢使用subplot的实例方法（因为我喜欢明确的事情，而且在处理多个subplot时这样也更清楚一些）。当然你完全可以选择自己觉得方便的那个。

设置标题、轴标签、刻度以及刻度标签

为了说明自定义轴，我将创建一个简单的图像并绘制一段随机漫步（如图9-8所示）：

```
In [37]: fig = plt.figure()

In [38]: ax = fig.add_subplot(1, 1, 1)

In [39]: ax.plot(np.random.randn(1000).cumsum())
```

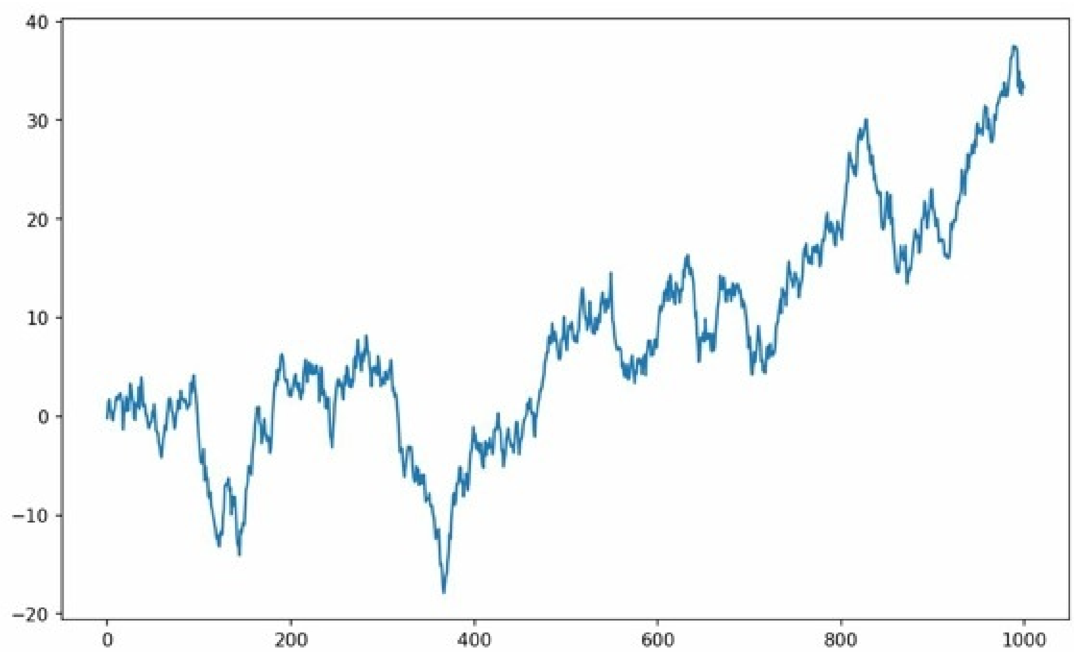


图9-8 用于演示xticks的简单线型图（带有标签）

要改变x轴刻度，最简单的办法是使用set_xticks和set_xticklabels。前者告诉matplotlib要将刻度放在数据范围中的哪些位置，默认情况下，这些位置也就是刻度标签。但我们可以通过set_xticklabels将任何其他的值用作标签：

```
In [40]: ticks = ax.set_xticks([0, 250, 500, 750, 100

In [41]: labels = ax.set_xticklabels(['one', 'two', '
   ....:                             rotation=30, fon
```

rotation选项设定x刻度标签倾斜30度。最后，再用set_xlabel为X轴设置一个名称，并用set_title设置一个标题（见图9-9的结果）：

```
In [42]: ax.set_title('My first matplotlib plot')
Out[42]: <matplotlib.text.Text at 0x7fb624d055f8>

In [43]: ax.set_xlabel('Stages')
```

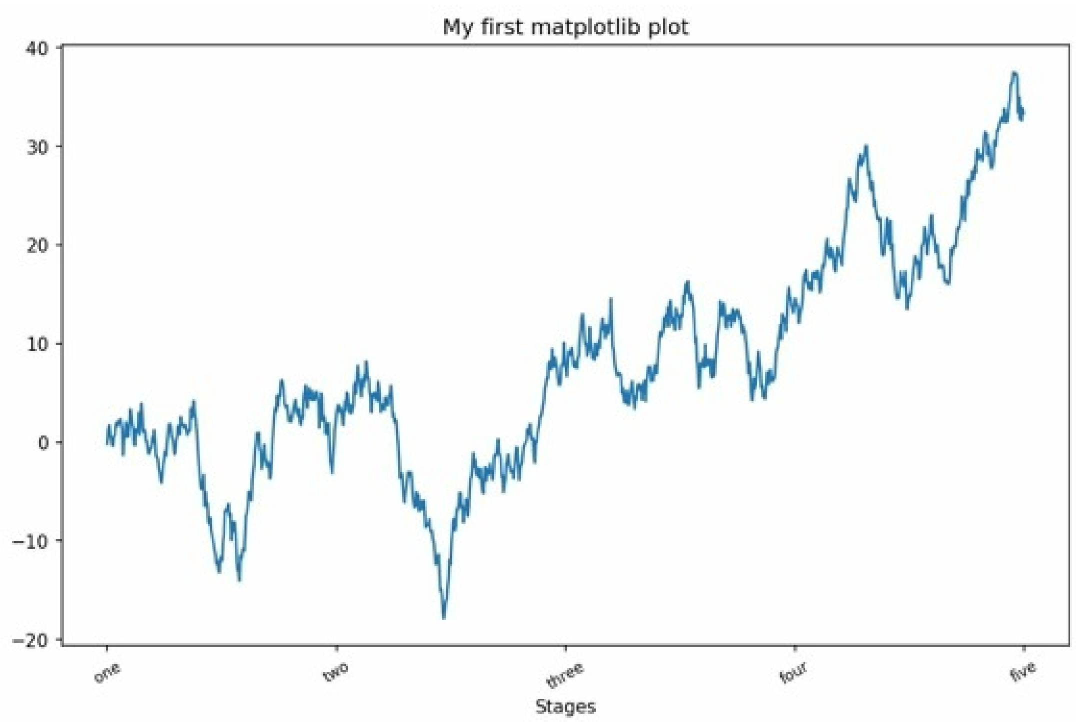


图9-9 用于演示xticks的简单线型图

Y轴的修改方式与此类似，只需将上述代码中的x替换为y即可。轴的类有集合方法，可以批量设定绘图选项。前面的例子，也可以写为：

```
props = {
    'title': 'My first matplotlib plot',
    'xlabel': 'Stages'
}
ax.set(**props)
```

添加图例

图例（legend）是另一种用于标识图表元素的重要工具。添加图例的方式有多种。最简单的是在添加subplot的时候传入label参数：

```
In [44]: from numpy.random import randn

In [45]: fig = plt.figure(); ax = fig.add_subplot(1,
```

```
In [46]: ax.plot(randn(1000).cumsum(), 'k', label='on
Out[46]: [<matplotlib.lines.Line2D at 0x7fb624bdf860>

In [47]: ax.plot(randn(1000).cumsum(), 'k--', label='
Out[47]: [<matplotlib.lines.Line2D at 0x7fb624be90f0>

In [48]: ax.plot(randn(1000).cumsum(), 'k.', label='t
Out[48]: [<matplotlib.lines.Line2D at 0x7fb624be9160>
```

在此之后，你可以调用ax.legend()或plt.legend()来自动创建图例（结果见图9-10）：

```
In [49]: ax.legend(loc='best')
```

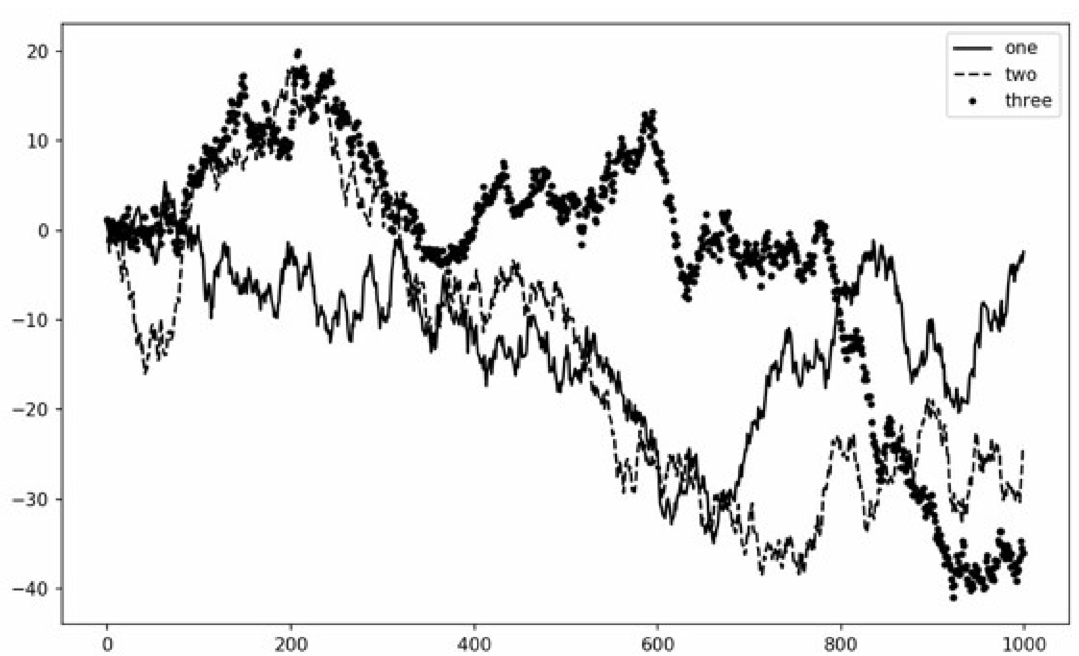


图9-10 带有三条线以及图例的简单线型图

legend方法有几个其它的loc位置参数选项。请查看文档字符串（使用ax.legend?）。

loc告诉matplotlib要将图例放在哪。如果你不是吹毛求疵的话，"best"是不错的选择，因为它会选择最不碍事的位置。要从

图例中去除一个或多个元素，不传入label或传入label='*nolegend*'即可。（中文第一版这里把best错写成了beat）

注解以及在Subplot上绘图

除标准的绘图类型，你可能还希望绘制一些子集的注解，可能是文本、箭头或其他图形等。注解和文字可以通过text、arrow和annotate函数进行添加。text可以将文本绘制在图表的指定坐标(x,y)，还可以加上一些自定义格式：

```
ax.text(x, y, 'Hello world!',
        family='monospace', fontsize=10)
```

注解中可以既含有文本也含有箭头。例如，我们根据最近的标准普尔500指数价格（来自Yahoo!Finance）绘制一张曲线图，并标出2008年到2009年金融危机期间的一些重要日期。你可以在Jupyter notebook的一个小窗中试验这段代码（图9-11是结果）：

```
from datetime import datetime

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

data = pd.read_csv('examples/spx.csv', index_col=0, pa
spx = data['SPX']

spx.plot(ax=ax, style='k-')

crisis_data = [
    (datetime(2007, 10, 11), 'Peak of bull market'),
    (datetime(2008, 3, 12), 'Bear Stearns Fails'),
    (datetime(2008, 9, 15), 'Lehman Bankruptcy')
]

for date, label in crisis_data:
```

```
    ax.annotate(label, xy=(date, spx.asof(date) + 75)
                xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(facecolor='black', he
                                headlength=4),
                horizontalalignment='left', verticala

# Zoom in on 2007-2010
ax.set_xlim(['1/1/2007', '1/1/2011'])
ax.set_ylim([600, 1800])

ax.set_title('Important dates in the 2008-2009 financ
```

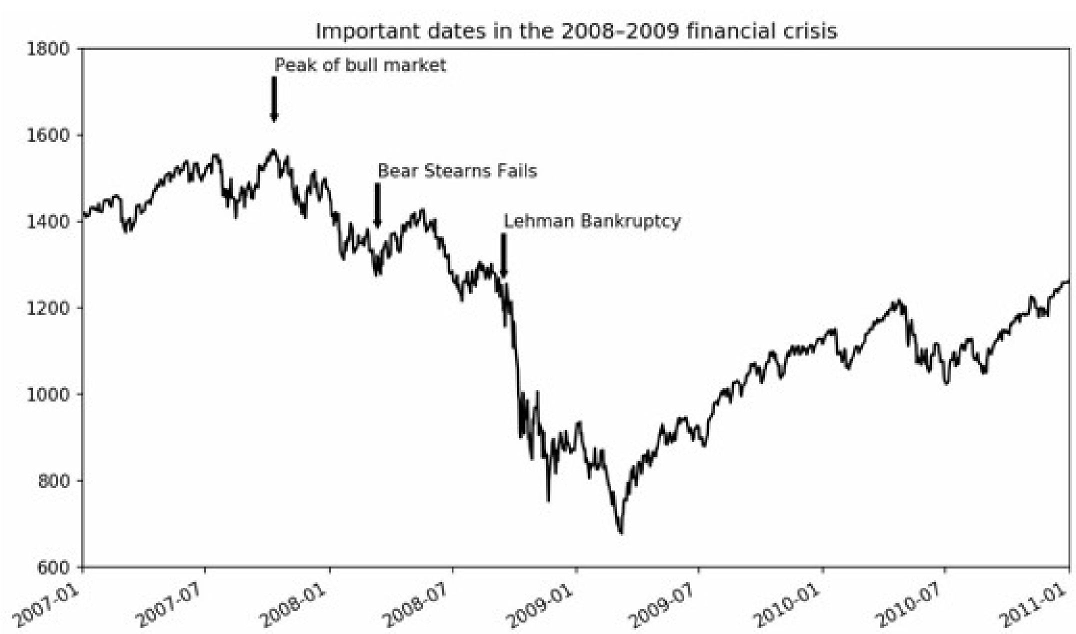


图9-11 2008-2009年金融危机期间的重要日期

这张图中有几个重要的点要强调：ax.annotate方法可以在指定的x和y坐标轴绘制标签。我们使用set_xlim和set_ylim人工设定起始和结束边界，而不使用matplotlib的默认方法。最后，用ax.set_title添加图标标题。

更多有关注解的示例，请访问matplotlib的在线示例库。

图形的绘制要麻烦一些。matplotlib有一些表示常见图形的对象。这些对象被称为块（patch）。其中有些（如Rectangle和

Circle），可以在matplotlib.pyplot中找到，但完整集合位于matplotlib.patches。

要在图表中添加一个图形，你需要创建一个块对象shp，然后通过ax.add_patch(shp)将其添加到subplot中（如图9-12所示）：

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=(
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, (
                   color='g', alpha=0.5)

ax.add_patch(rect)
ax.add_patch(circ)
ax.add_patch(pgon)
```

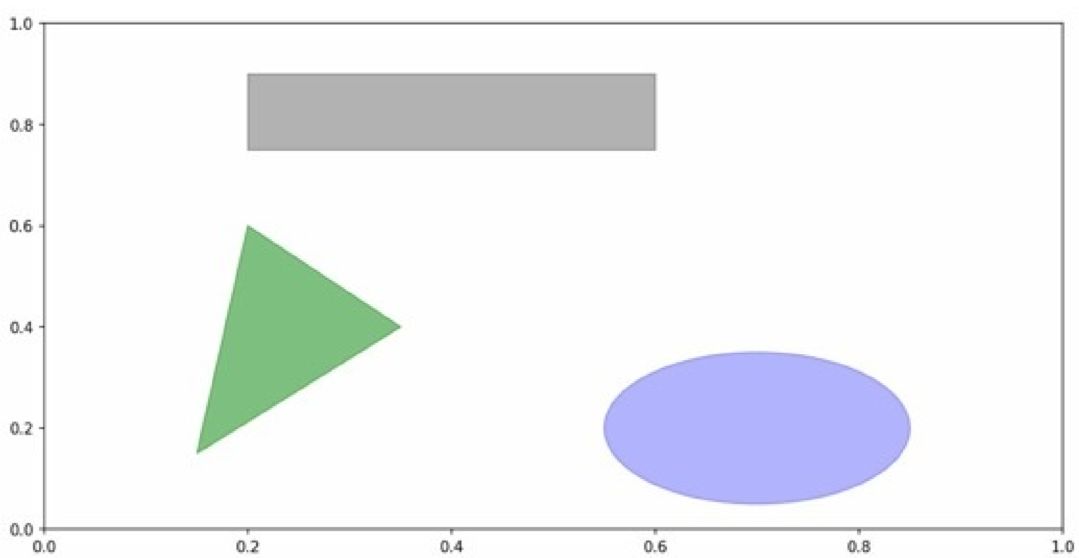


图9-12 由三个块图形组成的图

如果查看许多常见图表对象的具体实现代码，你就会发现它们其实就是由块patch组装而成的。

将图表保存到文件

利用plt.savefig可以将当前图表保存到文件。该方法相当于Figure对象的实例方法savefig。例如，要将图表保存为SVG文件，你只需输入：

```
plt.savefig('figpath.svg')
```

文件类型是通过文件扩展名推断出来的。因此，如果你使用的是.pdf，就会得到一个PDF文件。我在发布图片时最常用到两个重要的选项是dpi（控制“每英寸点数”分辨率）和bbox_inches（可以剪除当前图表周围的空白部分）。要得到一张带有最小白边且分辨率为400DPI的PNG图片，你可以：

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tigh
```

savefig并非一定要写入磁盘，也可以写入任何文件型的对象，比如BytesIO：

```
from io import BytesIO
buffer = BytesIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

表9-2列出了savefig的其它选项。

参数	说明
fname	含有文件路径的字符串或Python的文件型对象。图像格式由文件扩展名推断得出，例如，.pdf推断出PDF，.png推断出PNG
dpi	图像分辨率（每英寸点数），默认为100
facecolor、edgecolor	图像的背景色，默认为“w”（白色）
format	显式设置文件格式（“png”、“pdf”、“svg”、“ps”、“eps”……）
bbox_inches	图表需要保存的部分。如果设置为“tight”，则将尝试剪除图表周围的空白部分

表9-2 Figure.savefig的选项

matplotlib配置

matplotlib自带一些配色方案，以及为生成出版质量的图片而设定的默认配置信息。幸运的是，几乎所有默认行为都能通过一组全局参数进行自定义，它们可以管理图像大小、subplot边距、配色方案、字体大小、网格类型等。一种Python编程方式配置系统的方法是使用rc方法。例如，要将全局的图像默认大小设置为10×10，你可以执行：

```
plt.rc('figure', figsize=(10, 10))
```

rc的第一个参数是希望自定义的对象，如'figure'、'axes'、'xtick'、'ytick'、'grid'、'legend'等。其后可以跟上一系列的关键字参数。一个简单的办法是将这些选项写成一个字典：

```
font_options = {'family' : 'monospace',
                'weight' : 'bold',
                'size'   : 'small'}
plt.rc('font', **font_options)
```

要了解全部的自定义选项，请查阅matplotlib的配置文件matplotlibrc（位于matplotlib/mpl-data目录中）。如果对该文件进行了自定义，并将其放在你自己的.matplotlibrc目录中，则每次使用matplotlib时就会加载该文件。

下一节，我们会看到，seaborn包有若干内置的绘图主题或类型，它们使用了matplotlib的内部配置。

9.2 使用pandas和seaborn绘图

matplotlib实际上是一种比较低级的工具。要绘制一张图表，你组装一些基本组件就行：数据展示（即图表类型：线型图、柱状图、盒形图、散布图、等值线图等）、图例、标题、刻度标签以及其他注解型信息。

在pandas中，我们有多列数据，还有行和列标签。pandas自身就有内置的方法，用于简化从DataFrame和Series绘制图形。另

一个库seaborn（https://seaborn.pydata.org/），由Michael Waskom创建的静态图形库。Seaborn简化了许多常见可视类型的创建。

> 提示：引入seaborn会修改matplotlib默认的颜色方案和绘图类型，以提高可读性和美观度。即使你不使用seaborn API，你可能也会引入seaborn，作为提高美观度和绘制常见matplotlib图形的简化方法。

线型图

Series和DataFrame都有一个用于生成各类图表的plot方法。默认情况下，它们所生成的是线型图（如图9-13所示）：

```
In [60]: s = pd.Series(np.random.randn(10).cumsum(), 

In [61]: s.plot()
```

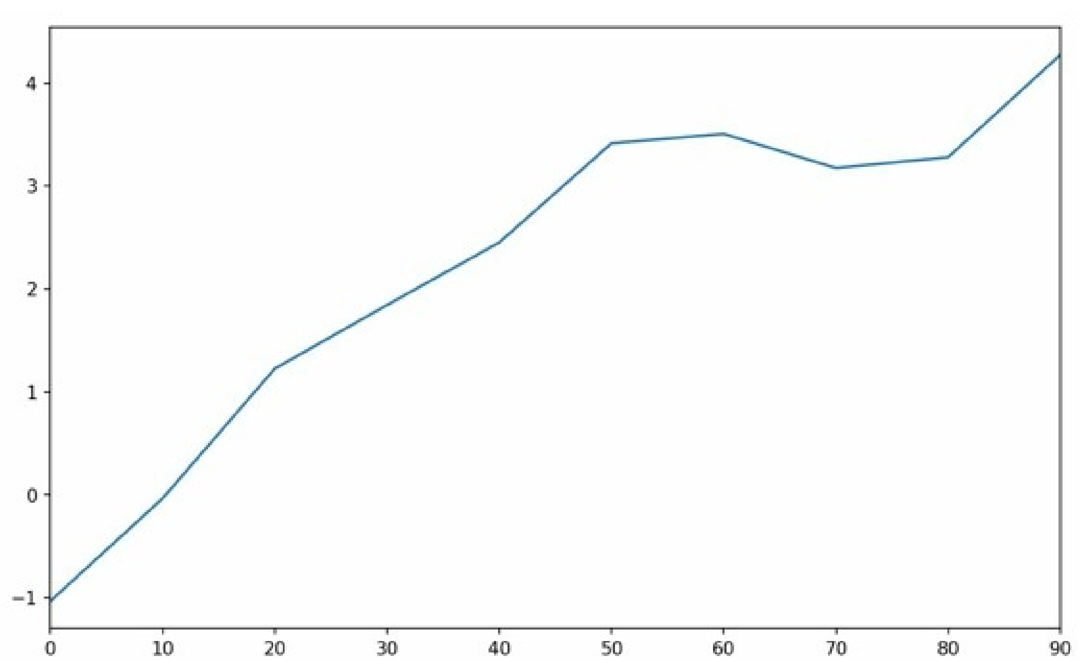


图9-13 简单的Series图表示例

该Series对象的索引会被传给matplotlib，并用以绘制X轴。可以通过use_index=False禁用该功能。X轴的刻度和界限可以通过

xticks和xlim选项进行调节，Y轴就用yticks和ylim。plot参数的完整列表请参见表9-3。我只会讲解其中几个，剩下的就留给读者自己去研究了。

参数	说明
label	用于图例的标签
ax	要在其上进行绘制的matplotlib subplot对象。如果没有设置，则使用当前matplotlib subplot
style	将要传给matplotlib的风格字符串（如'ko--'）
alpha	图表的填充不透明度（0到1之间）
kind	可以是'line'、'bar'、'barh'、'kde'
logy	在Y轴上使用对数标尺
use_index	将对象的索引用作刻度标签
rot	旋转刻度标签（0到360）
xticks	用作X轴刻度的值
yticks	用作Y轴刻度的值
xlim	X轴的界限（例如[0, 10]）
ylim	Y轴的界限
grid	显示轴网格线（默认打开）

表9-3 Series.plot方法的参数

pandas的大部分绘图方法都有一个可选的ax参数，它可以是一个matplotlib的subplot对象。这使你能够在网格布局中更为灵活地处理subplot的位置。

DataFrame的plot方法会在一个subplot中为各列绘制一条线，并自动创建图例（如图9-14所示）：

```
In [62]: df = pd.DataFrame(np.random.randn(10, 4).cum
   ....:                   columns=['A', 'B', 'C', 'D
   ....:                   index=np.arange(0, 100, 10

In [63]: df.plot()
```

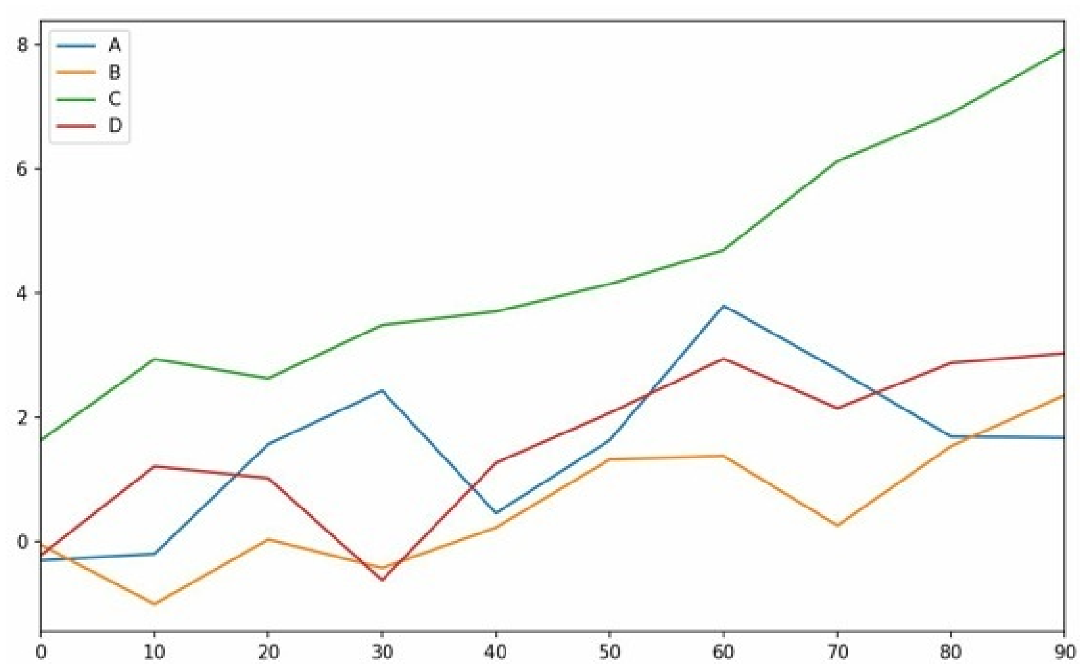


图9-14 简单的DataFrame绘图

plot属性包含一批不同绘图类型的方法。例如，df.plot()等价于df.plot.line()。后面会学习这些方法。

> 笔记：plot的其他关键字参数会被传给相应的matplotlib绘图函数，所以要更深入地自定义图表，就必须学习更多有关matplotlib API的知识。

DataFrame还有一些用于对列进行灵活处理的选项，例如，是要将所有列都绘制到一个subplot中还是创建各自的subplot。详细信息请参见表9-4。

参数	说明
subplots	将各个DataFrame列绘制到单独的subplot中
sharex	如果subplots=True，则共用同一个X轴，包括刻度和界限
sharey	如果subplots=True，则共用同一个Y轴
figsize	表示图像大小的元组
title	表示图像标题的字符串
legend	添加一个subplot图例（默认为True）
sort_columns	以字母表顺序绘制各列，默认使用当前列顺序

表9-4 专用于DataFrame的plot参数

> 注意： 有关时间序列的绘图，请见第11章。

柱状图

plot.bar()和plot.barh()分别绘制水平和垂直的柱状图。这时，Series和DataFrame的索引将会被用作X（bar）或Y（barh）刻度（如图9-15所示）：

```
In [64]: fig, axes = plt.subplots(2, 1)

In [65]: data = pd.Series(np.random.rand(16), index=l

In [66]: data.plot.bar(ax=axes[0], color='k', alpha=0
Out[66]: <matplotlib.axes._subplots.AxesSubplot at 0x

In [67]: data.plot.barh(ax=axes[1], color='k', alpha=
```

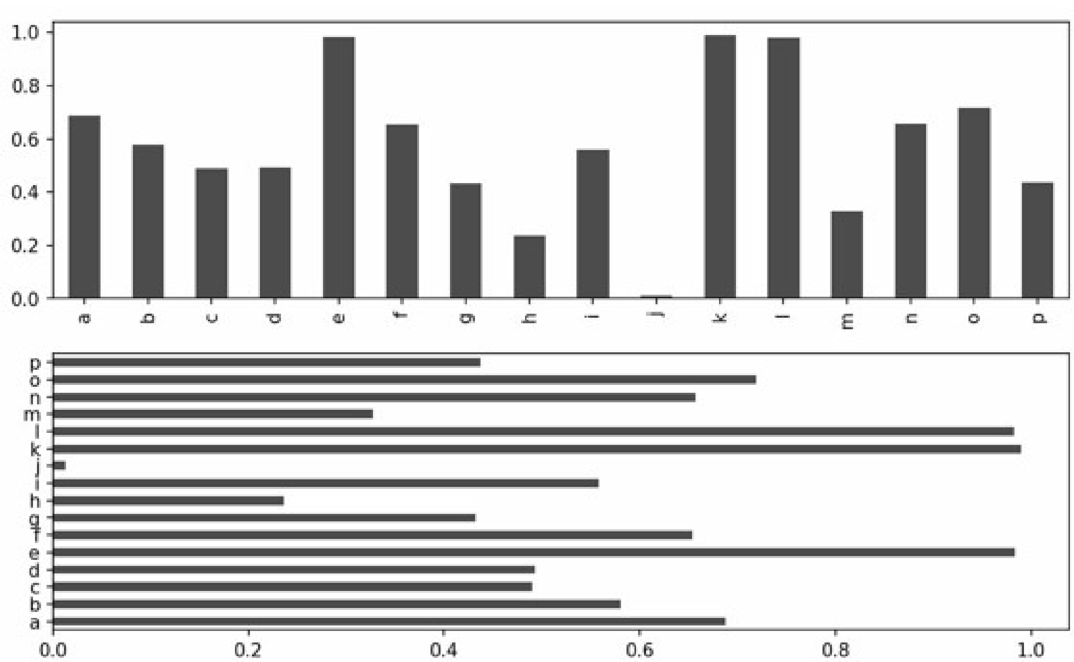


图9-15 水平和垂直的柱状图

color='k'和alpha=0.7设定了图形的颜色为黑色，并使用部分的填充透明度。对于DataFrame，柱状图会将每一行的值分为一组，并排显示，如图9-16所示：

```
In [69]: df = pd.DataFrame(np.random.rand(6, 4),
   ....:                   index=['one', 'two', 'thre
   ....:                   columns=pd.Index(['A', 'B'

In [70]: df
Out[70]:
Genus         A         B         C         D
one    0.370670  0.602792  0.229159  0.486744
two    0.420082  0.571653  0.049024  0.880592
three  0.814568  0.277160  0.880316  0.431326
four   0.374020  0.899420  0.460304  0.100843
five   0.433270  0.125107  0.494675  0.961825
six    0.601648  0.478576  0.205690  0.560547

In [71]: df.plot.bar()
```

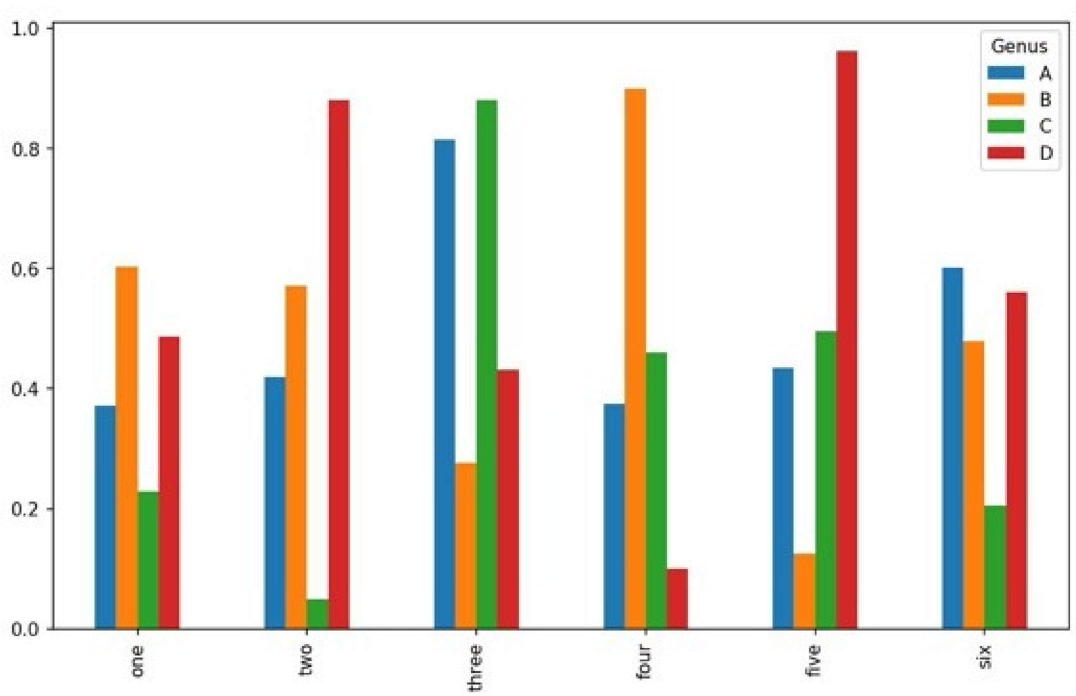


图9-16 DataFrame的柱状图

注意，DataFrame各列的名称"Genus"被用作了图例的标题。

设置stacked=True即可为DataFrame生成堆积柱状图，这样每行

的值就会被堆积在一起（如图9-17所示）：

```
In [73]: df.plot.barh(stacked=True, alpha=0.5)
```

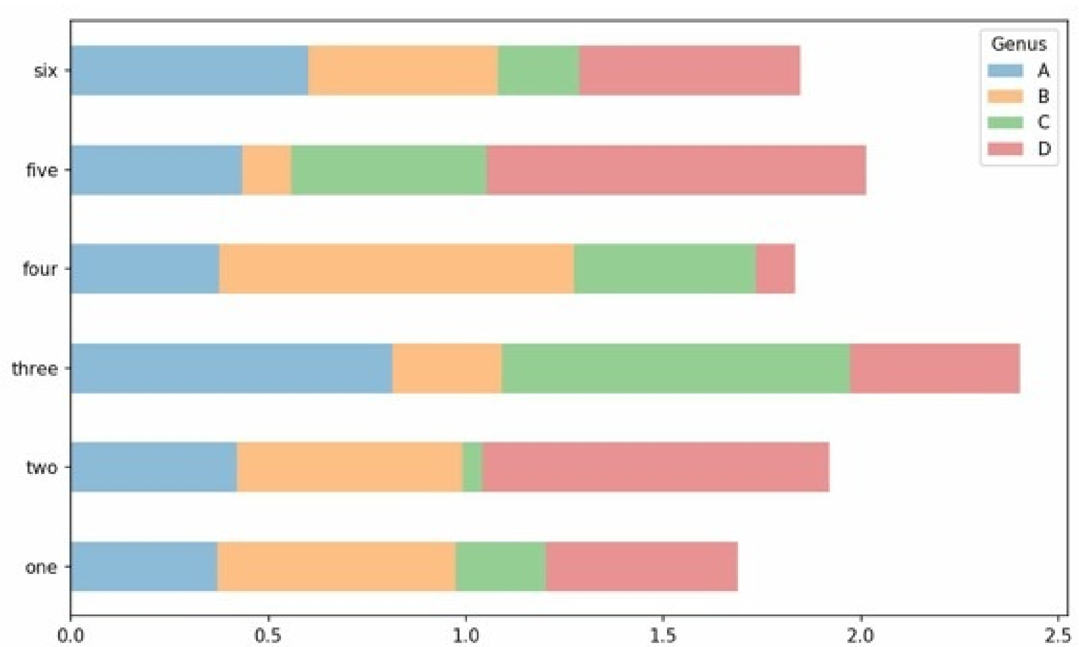


图9-17 DataFrame的堆积柱状图

> 笔记：柱状图有一个非常不错的用法：利用value_counts图形化显示Series中各值的出现频率，比如s.value_counts().plot.bar()。

再以本书前面用过的那个有关小费的数据集为例，假设我们想要做一张堆积柱状图以展示每天各种聚会规模的数据点的百分比。我用read_csv将数据加载进来，然后根据日期和聚会规模创建一张交叉表：

```
In [75]: tips = pd.read_csv('examples/tips.csv')

In [76]: party_counts = pd.crosstab(tips['day'], tips

In [77]: party_counts
Out[77]:
size  1   2   3   4  5  6
day
```

```
Fri    1  16   1   1  0  0
Sat    2  53  18  13  1  0
Sun    0  39  15  18  3  1
Thur   1  48   4   5  1  3

# Not many 1- and 6-person parties
In [78]: party_counts = party_counts.loc[:, 2:5]
```

然后进行规格化，使得各行的和为1，并生成图表（如图9-18所示）：

```
# Normalize to sum to 1
In [79]: party_pcts = party_counts.div(party_counts.s

In [80]: party_pcts
Out[80]:
size         2         3         4         5
day
Fri   0.888889  0.055556  0.055556  0.000000
Sat   0.623529  0.211765  0.152941  0.011765
Sun   0.520000  0.200000  0.240000  0.040000
Thur  0.827586  0.068966  0.086207  0.017241

In [81]: party_pcts.plot.bar()
```

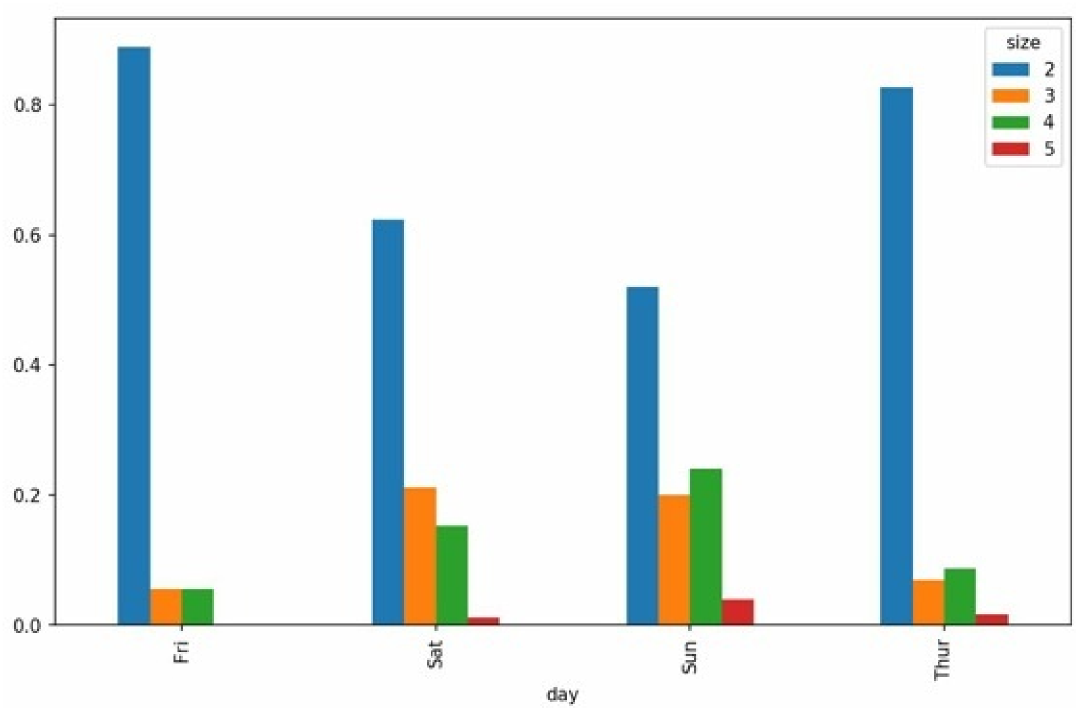


图9-18 每天各种聚会规模的比例

于是，通过该数据集就可以看出，聚会规模在周末会变大。

对于在绘制一个图形之前，需要进行合计的数据，使用seaborn可以减少工作量。用seaborn来看每天的小费比例（图9-19是结果）：

```
In [83]: import seaborn as sns

In [84]: tips['tip_pct'] = tips['tip'] / (tips['total_

In [85]: tips.head()
Out[85]:
   total_bill   tip smoker  day    time  size   tip_p
0       16.99  1.01     No  Sun  Dinner     2  0.0632
1       10.34  1.66     No  Sun  Dinner     3  0.1912
2       21.01  3.50     No  Sun  Dinner     3  0.1998
3       23.68  3.31     No  Sun  Dinner     2  0.1624
4       24.59  3.61     No  Sun  Dinner     4  0.1720

In [86]: sns.barplot(x='tip_pct', y='day', data=tips,
```

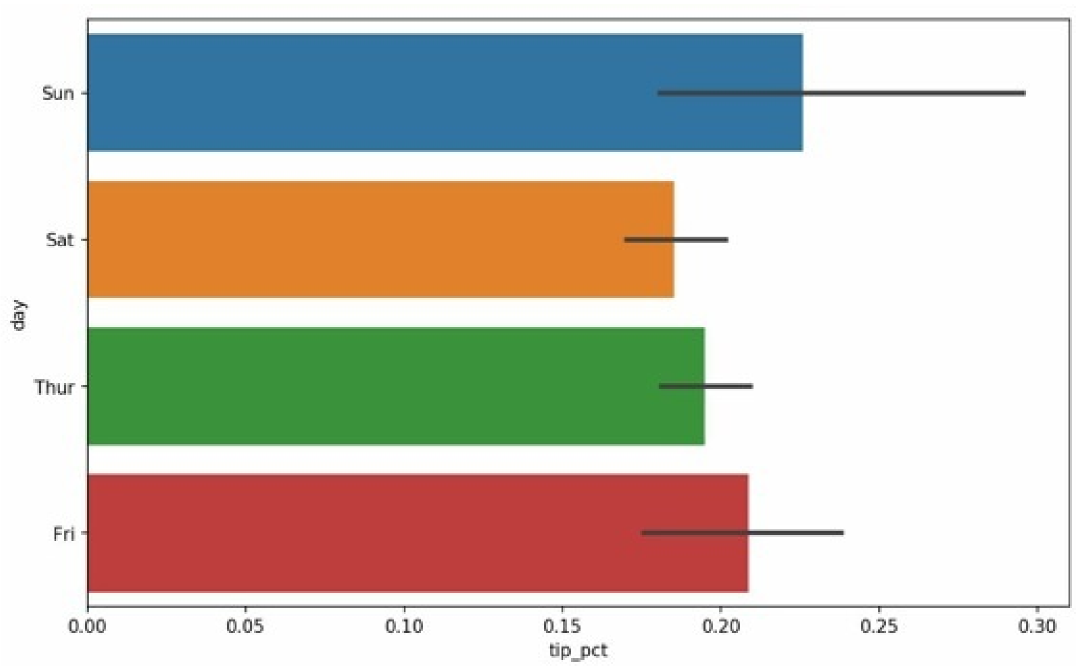


图9-19 小费的每日比例，带有误差条

seaborn的绘制函数使用data参数，它可能是pandas的DataFrame。其它的参数是关于列的名字。因为一天的每个值有多次观察，柱状图的值是tip_pct的平均值。绘制在柱状图上的黑线代表95%置信区间（可以通过可选参数配置）。

seaborn.barplot有颜色选项，使我们能够通过一个额外的值设置（见图9-20）：

```
In [88]: sns.barplot(x='tip_pct', y='day', hue='time'
```

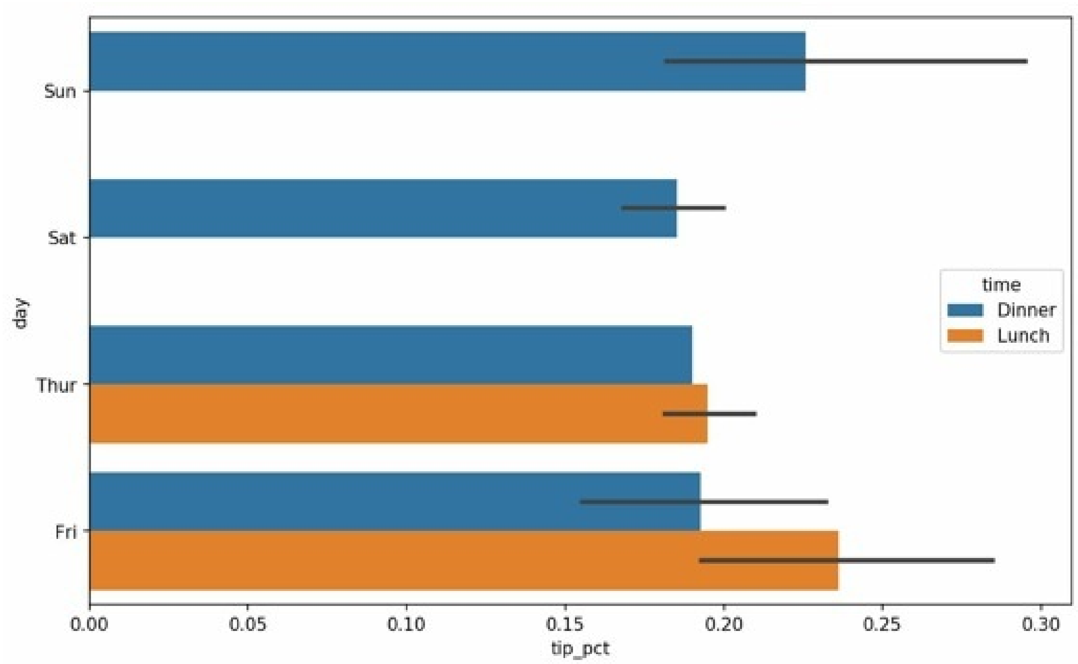


图9-20 根据天和时间的小费比例

注意，seaborn已经自动修改了图形的美观度：默认调色板，图形背景和网格线的颜色。你可以用seaborn.set在不同的图形外观之间切换：

```
In [90]: sns.set(style="whitegrid")
```

直方图和密度图

直方图（histogram）是一种可以对值频率进行离散化显示的柱状图。数据点被拆分到离散的、间隔均匀的面元中，绘制的是各面元中数据点的数量。再以前面那个小费数据为例，通过在Series使用plot.hist方法，我们可以生成一张“小费占消费总额百分比”的直方图（如图9-21所示）：

```
In [92]: tips['tip_pct'].plot.hist(bins=50)
```

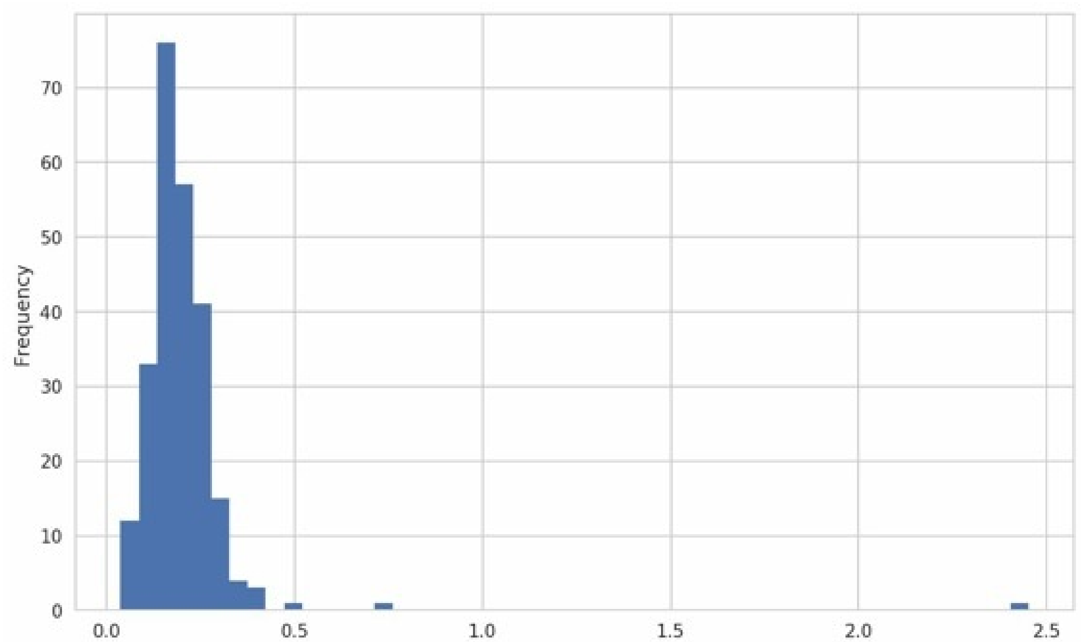


图9-21 小费百分比的直方图

与此相关的一种图表类型是密度图，它是通过计算“可能会产生观测数据的连续概率分布的估计”而产生的。一般的过程是将该分布近似为一组核（即诸如正态分布之类的较为简单的分布）。因此，密度图也被称作KDE（Kernel Density Estimate，核密度估计）图。使用plot.kde和标准混合正态分布估计即可生成一张密度图（见图9-22）：

```
In [94]: tips['tip_pct'].plot.density()
```

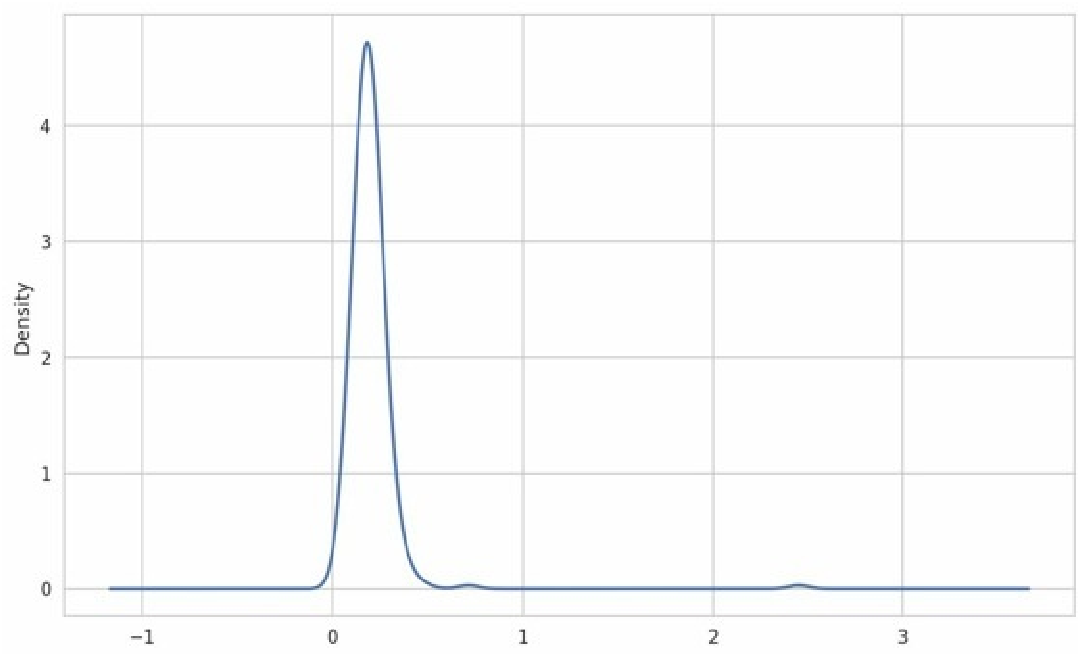


图9-22 小费百分比的密度图

seaborn的distplot方法绘制直方图和密度图更加简单，还可以同时画出直方图和连续密度估计图。作为例子，考虑一个双峰分布，由两个不同的标准正态分布组成（见图9-23）：

```
In [96]: comp1 = np.random.normal(0, 1, size=200)

In [97]: comp2 = np.random.normal(10, 2, size=200)

In [98]: values = pd.Series(np.concatenate([comp1, co

In [99]: sns.distplot(values, bins=100, color='k')
```

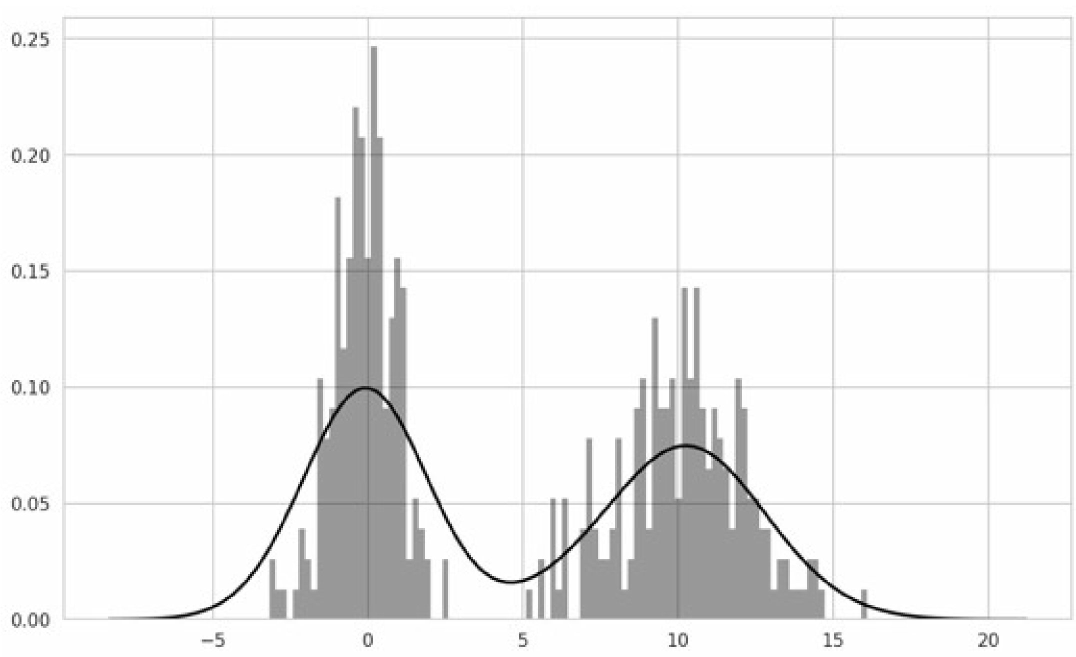


图9-23 标准混合密度估计的标准直方图

散布图或点图

点图或散布图是观察两个一维数据序列之间的关系的有效手段。在下面这个例子中，我加载了来自statsmodels项目的macrodata数据集，选择了几个变量，然后计算对数差：

```
In [100]: macro = pd.read_csv('examples/macrodata.csv

In [101]: data = macro[['cpi', 'm1', 'tbilrate', 'une

In [102]: trans_data = np.log(data).diff().dropna()

In [103]: trans_data[-5:]
Out[103]:
          cpi        m1  tbilrate     unemp
198 -0.007904  0.045361 -0.396881  0.105361
199 -0.021979  0.066753 -2.277267  0.139762
200  0.002340  0.010286  0.606136  0.160343
201  0.008419  0.037461 -0.200671  0.127339
202  0.008894  0.012202 -0.405465  0.042560
```

然后可以使用seaborn的regplot方法，它可以做一个散布图，并加上一条线性回归的线（见图9-24）：

```
In [105]: sns.regplot('m1', 'unemp', data=trans_data)
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x

In [106]: plt.title('Changes in log %s versus log %s'
```

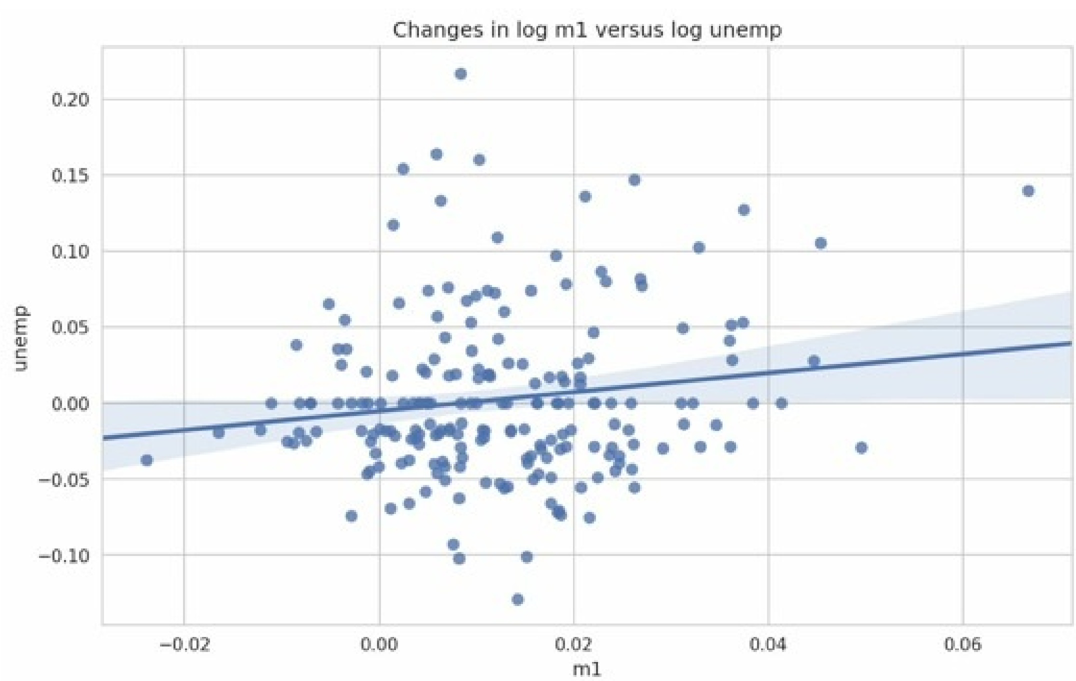


图9-24 seaborn的回归/散布图

在探索式数据分析工作中，同时观察一组变量的散布图是很有意义的，这也被称为散布图矩阵（scatter plot matrix）。纯手工创建这样的图表很费工夫，所以seaborn提供了一个便捷的pairplot函数，它支持在对角线上放置每个变量的直方图或密度估计（见图9-25）：

```
In [107]: sns.pairplot(trans_data, diag_kind='kde', p
```

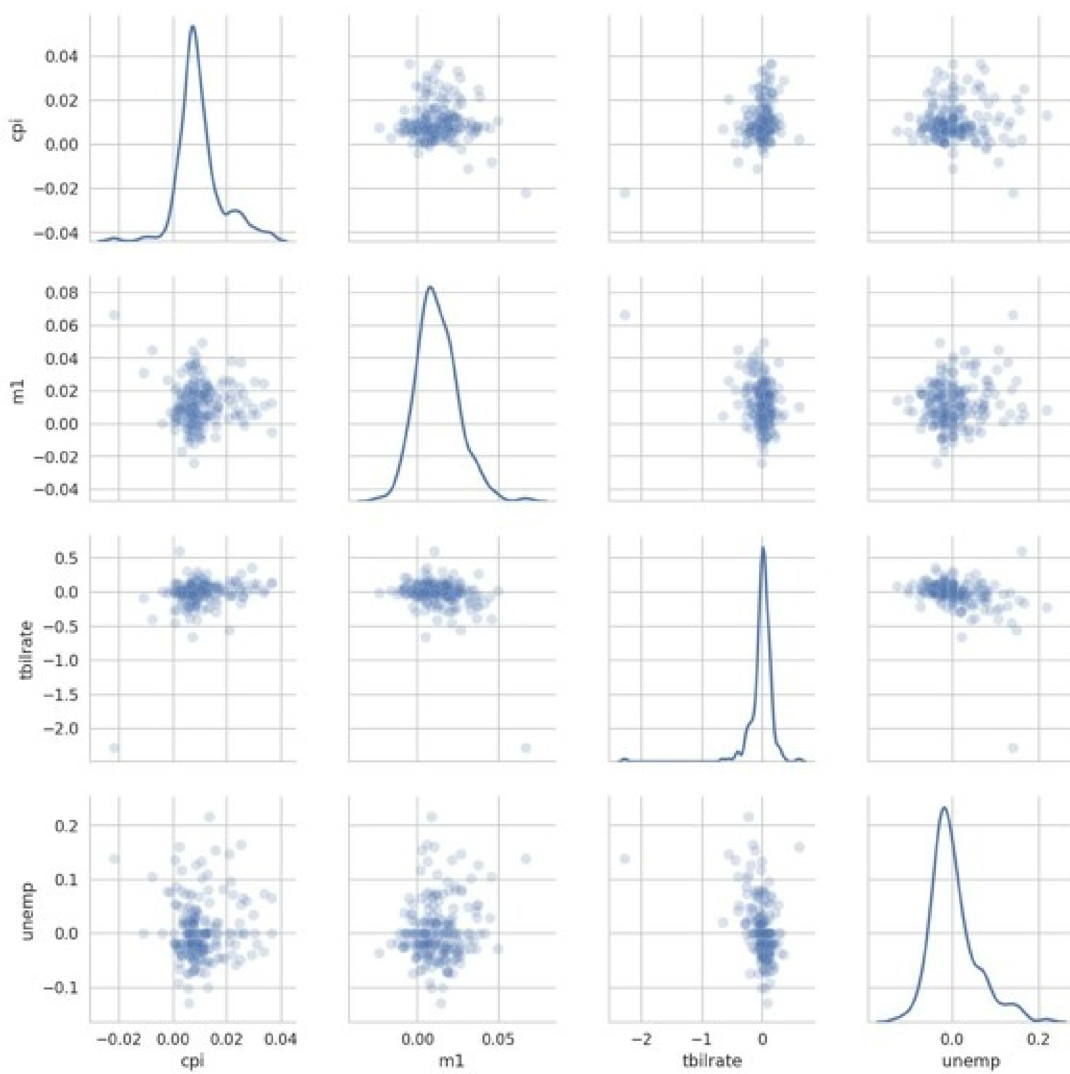


图9-25 statsmodels macro data的散布图矩阵

你可能注意到了plot_kws参数。它可以让我们传递配置选项到非对角线元素上的图形使用。对于更详细的配置选项，可以查阅seaborn.pairplot文档字符串。

分面网格（facet grid）和类型数据

要是数据集有额外的分组维度呢？有多个分类变量的数据可视化的一种方法是使用小面网格。seaborn有一个有用的内置函数factorplot，可以简化制作多种分面图（见图9-26）：

```
In [108]: sns.factorplot(x='day', y='tip_pct', hue='
   .....:                kind='bar', data=tips[tips.t
```

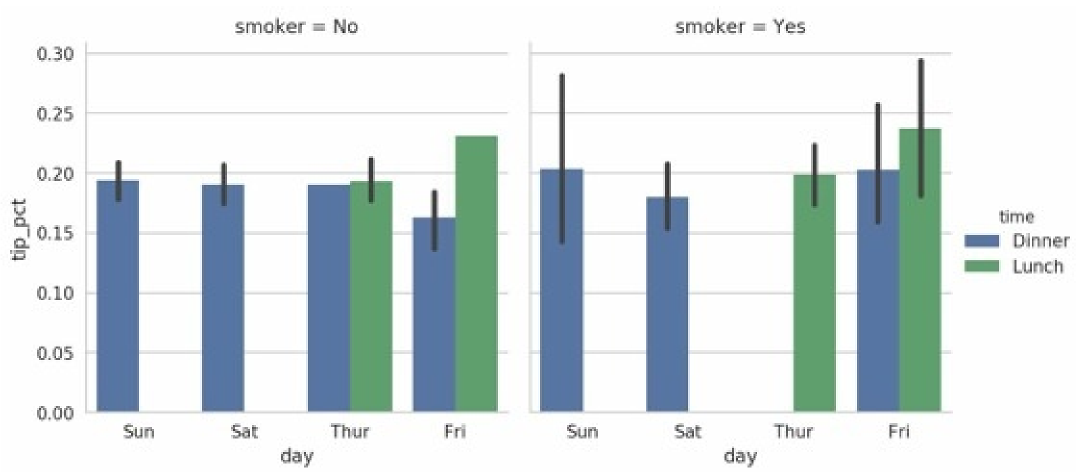


图9-26 按照天/时间/吸烟者的小费百分比

除了在分面中用不同的颜色按时间分组，我们还可以通过给每个时间值添加一行来扩展分面网格：

```
In [109]: sns.factorplot(x='day', y='tip_pct', row='t
   .....:                col='smoker',
   .....:                kind='bar', data=tips[tips.t
```

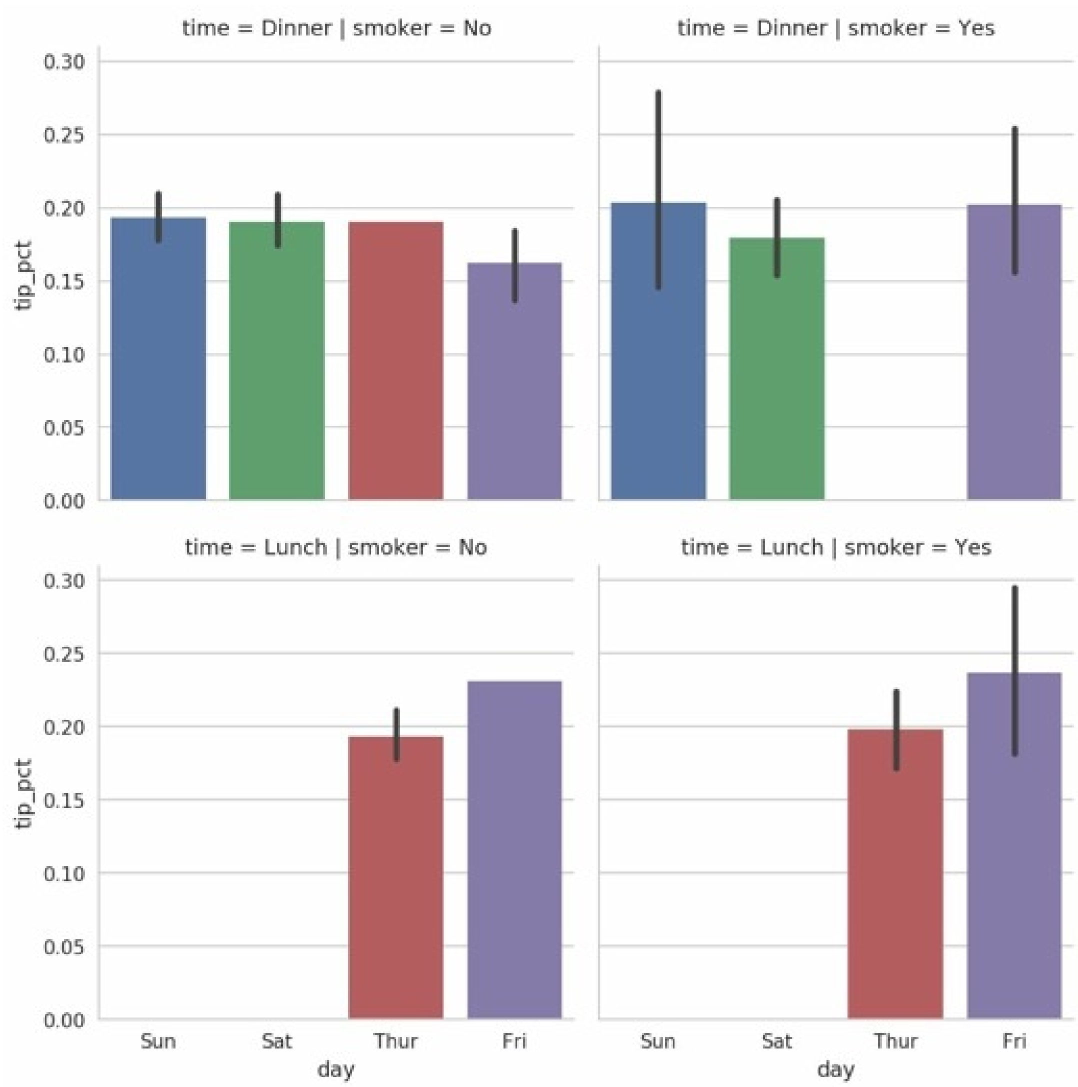


图9-27 按天的tip_pct，通过time/smoker分面

factorplot支持其它的绘图类型，你可能会用到。例如，盒图（它可以显示中位数，四分位数，和异常值）就是一个有用的可视化类型（见图9-28）：

```
In [110]: sns.factorplot(x='tip_pct', y='day', kind='
   .....:                 data=tips[tips.tip_pct < 0.5
```

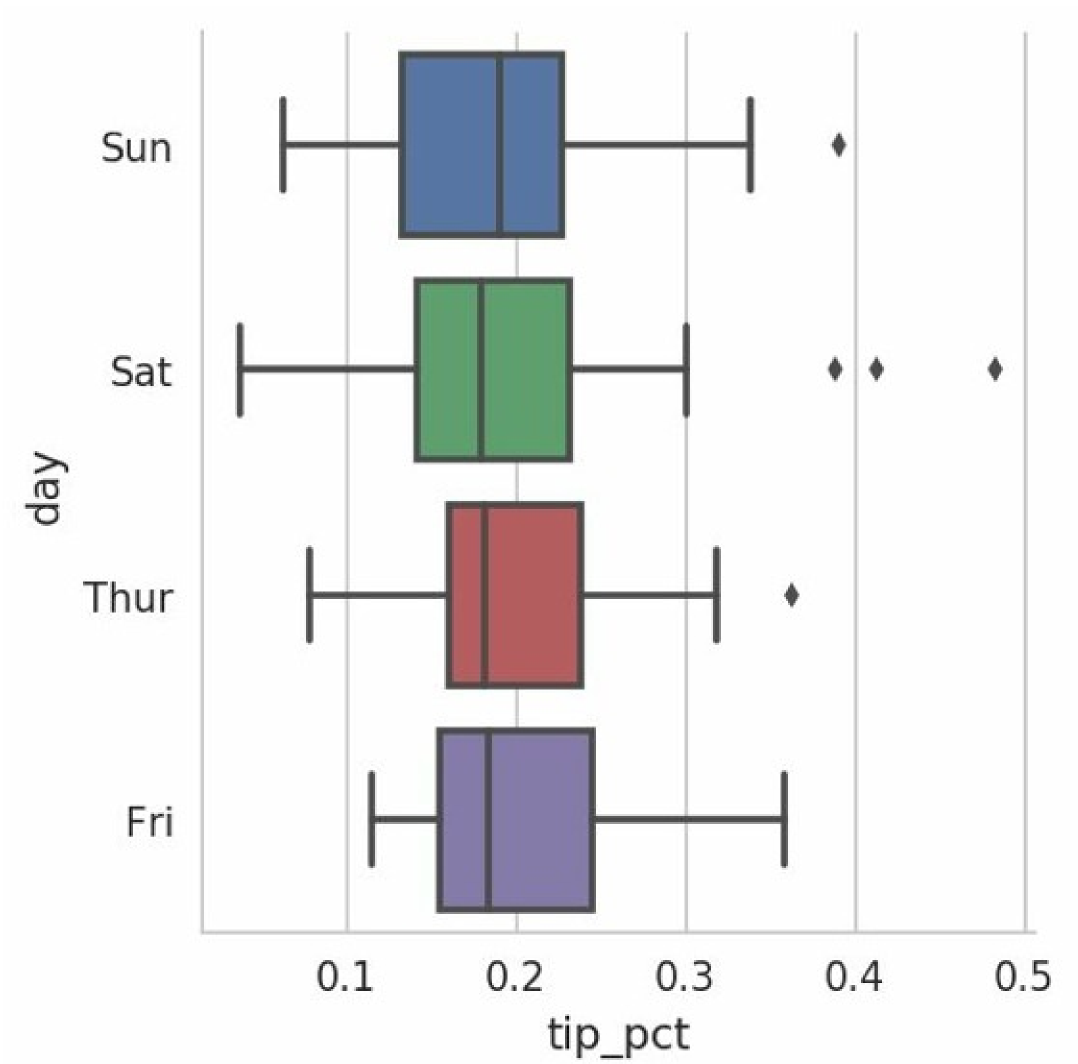


图9-28 按天的tip_pct的盒图

使用更通用的seaborn.FacetGrid类，你可以创建自己的分面网格。请查阅seaborn的文档（https://seaborn.pydata.org/）。

9.3 其它的Python可视化工具

与其它开源库类似，Python创建图形的方式非常多（根本罗列不完）。自从2010年，许多开发工作都集中在创建交互式图形以便在Web上发布。利用工具如
Boken（https://bokeh.pydata.org/en/latest/）和
Plotly（https://github.com/plotly/plotly.py），现在可以创建动态交互图形，用于网页浏览器。

对于创建用于打印或网页的静态图形，我建议默认使用matplotlib和附加的库，比如pandas和seaborn。对于其它数据可视化要求，学习其它的可用工具可能是有用的。我鼓励你探索绘图的生态系统，因为它将持续发展。

9.4 总结

本章的目的是熟悉一些基本的数据可视化操作，使用pandas，matplotlib，和seaborn。如果视觉显示数据分析的结果对你的工作很重要，我鼓励你寻求更多的资源来了解更高效的数据可视化。这是一个活跃的研究领域，你可以通过在线和纸质的形式学习许多优秀的资源。

下一章，我们将重点放在pandas的数据聚合和分组操作上。

第10章 数据聚合与分组运算

对数据集进行分组并对各组应用一个函数（无论是聚合还是转换），通常是数据分析工作中的重要环节。在将数据集加载、融合、准备好之后，通常就是计算分组统计或生成透视表。pandas提供了一个灵活高效的gruopby功能，它使你能以一种自然的方式对数据集进行切片、切块、摘要等操作。

关系型数据库和SQL（Structured Query Language，结构化查询语言）能够如此流行的原因之一就是其能够方便地对数据进行连接、过滤、转换和聚合。但是，像SQL这样的查询语言所能执行的分组运算的种类很有限。在本章中你将会看到，由于Python和pandas强大的表达能力，我们可以执行复杂得多的分组运算（利用任何可以接受pandas对象或NumPy数组的函数）。在本章中，你将会学到：

- 计算分组摘要统计，如计数、平均值、标准差，或用户自定义函数。
- 计算分组的概述统计，比如数量、平均值或标准差，或是用户定义的函数。
- 应用组内转换或其他运算，如规格化、线性回归、排名或选取子集等。
- 计算透视表或交叉表。
- 执行分位数分析以及其它统计分组分析。

> 笔记：对时间序列数据的聚合（groupby的特殊用法之一）也称作重采样（resampling），本书将在第11章中单独对其进行讲解。

10.1 GroupBy机制

Hadley Wickham（许多热门R语言包的作者）创造了一个用于表示分组运算的术语"split-apply-combine"（拆分－应用－合并）。第一个阶段，pandas对象（无论是Series、DataFrame还是其他的）中的数据会根据你所提供的一个或多个键被拆分

（split）为多组。拆分操作是在对象的特定轴上执行的。例如，DataFrame可以在其行（axis=0）或列（axis=1）上进行分组。然后，将一个函数应用（apply）到各个分组并产生一个新值。最后，所有这些函数的执行结果会被合并（combine）到最终的结果对象中。结果对象的形式一般取决于数据上所执行的操作。图10-1大致说明了一个简单的分组聚合过程。

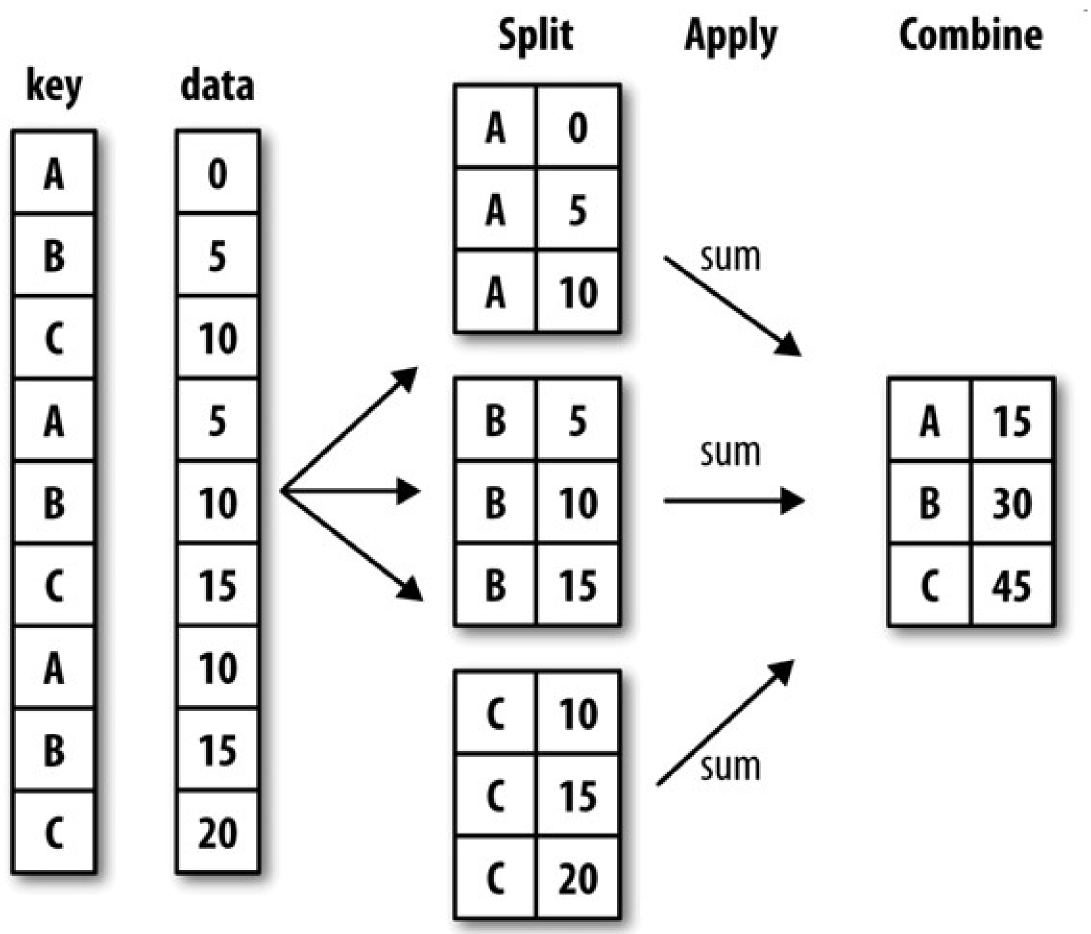


图10-1 分组聚合演示

分组键可以有多种形式，且类型不必相同：

- 列表或数组，其长度与待分组的轴一样。
- 表示DataFrame某个列名的值。
- 字典或Series，给出待分组轴上的值与分组名之间的对应关系。
- 函数，用于处理轴索引或索引中的各个标签。

注意，后三种都只是快捷方式而已，其最终目的仍然是产生一组用于拆分对象的值。如果觉得这些东西看起来很抽象，不用担心，我将在本章中给出大量有关于此的示例。首先来看看下面这个非常简单的表格型数据集（以DataFrame的形式）：

```
In [10]: df = pd.DataFrame({'key1' : ['a', 'a', 'b', 
   ....:                    'key2' : ['one', 'two', '
   ....:                    'data1' : np.random.randn
   ....:                    'data2' : np.random.randn

In [11]: df
Out[11]: 
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one
```

假设你想要按key1进行分组，并计算data1列的平均值。实现该功能的方式有很多，而我们这里要用的是：访问data1，并根据key1调用groupby：

```
In [12]: grouped = df['data1'].groupby(df['key1'])

In [13]: grouped
Out[13]: <pandas.core.groupby.SeriesGroupBy object at
```

变量grouped是一个GroupBy对象。它实际上还没有进行任何计算，只是含有一些有关分组键df['key1']的中间数据而已。换句话说，该对象已经有了接下来对各分组执行运算所需的一切信息。例如，我们可以调用GroupBy的mean方法来计算分组平均值：

```
In [14]: grouped.mean()
Out[14]:
```

```
key1
a    0.746672
b   -0.537585
Name: data1, dtype: float64
```

稍后我将详细讲解.mean()的调用过程。这里最重要的是，数据（Series）根据分组键进行了聚合，产生了一个新的Series，其索引为key1列中的唯一值。之所以结果中索引的名称为key1，是因为原始DataFrame的列df['key1']就叫这个名字。

如果我们一次传入多个数组的列表，就会得到不同的结果：

```
In [15]: means = df['data1'].groupby([df['key1'], df[

In [16]: means
Out[16]:
key1  key2
a     one     0.880536
      two     0.478943
b     one    -0.519439
      two    -0.555730
Name: data1, dtype: float64
```

这里，我通过两个键对数据进行了分组，得到的Series具有一个层次化索引（由唯一的键对组成）：

```
In [17]: means.unstack()
Out[17]:
key2       one       two
key1
a     0.880536  0.478943
b    -0.519439 -0.555730
```

在这个例子中，分组键均为Series。实际上，分组键可以是任何长度适当的数组：

```
In [18]: states = np.array(['Ohio', 'California', 'Ca

In [19]: years = np.array([2005, 2005, 2006, 2005, 20(

In [20]: df['data1'].groupby([states, years]).mean()
Out[20]:
California  2005    0.478943
            2006   -0.519439
Ohio        2005   -0.380219
            2006    1.965781
Name: data1, dtype: float64
```

通常，分组信息就位于相同的要处理DataFrame中。这里，你还可以将列名（可以是字符串、数字或其他Python对象）用作分组键：

```
In [21]: df.groupby('key1').mean()
Out[21]:
         data1     data2
key1
a     0.746672  0.910916
b    -0.537585  0.525384

In [22]: df.groupby(['key1', 'key2']).mean()
Out[22]:
              data1     data2
key1 key2
a    one   0.880536  1.319920
     two   0.478943  0.092908
b    one  -0.519439  0.281746
     two  -0.555730  0.769023
```

你可能已经注意到了，第一个例子在执行df.groupby('key1').mean()时，结果中没有key2列。这是因为df['key2']不是数值数据（俗称“麻烦列”），所以被从结果中排除

了。默认情况下，所有数值列都会被聚合，虽然有时可能会被过滤为一个子集，稍后就会碰到。

无论你准备拿groupby做什么，都有可能会用到GroupBy的size方法，它可以返回一个含有分组大小的Series：

```
In [23]: df.groupby(['key1', 'key2']).size()
Out[23]:
key1  key2
a     one     2
      two     1
b     one     1
      two     1
dtype: int64
```

注意，任何分组关键词中的缺失值，都会被从结果中除去。

对分组进行迭代

GroupBy对象支持迭代，可以产生一组二元元组（由分组名和数据块组成）。看下面的例子：

```
In [24]: for name, group in df.groupby('key1'):
   ....:     print(name)
   ....:     print(group)
   ....:
a
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
4  1.965781  1.246435    a  one
b
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

对于多重键的情况，元组的第一个元素将会是由键值组成的元

组：

```
In [25]: for (k1, k2), group in df.groupby(['key1', '
   ....:     print((k1, k2))
   ....:     print(group)
   ....:
('a', 'one')
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
4  1.965781  1.246435    a  one
('a', 'two')
      data1     data2 key1 key2
1  0.478943  0.092908    a  two
('b', 'one')
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
('b', 'two')
     data1     data2 key1 key2
3 -0.55573  0.769023    b  two
```

当然，你可以对这些数据片段做任何操作。有一个你可能会觉得有用的运算：将这些数据片段做成一个字典：

```
In [26]: pieces = dict(list(df.groupby('key1')))

In [27]: pieces['b']
Out[27]:
      data1     data2 key1 key2
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
```

groupby默认是在axis=0上进行分组的，通过设置也可以在其他任何轴上进行分组。拿上面例子中的df来说，我们可以根据dtype对列进行分组：

```
In [28]: df.dtypes
```

```
Out[28]:
data1    float64
data2    float64
key1      object
key2      object
dtype: object

In [29]: grouped = df.groupby(df.dtypes, axis=1)
```

可以如下打印分组：

```
In [30]: for dtype, group in grouped:
   ....:     print(dtype)
   ....:     print(group)
   ....:
float64
      data1     data2
0 -0.204708  1.393406
1  0.478943  0.092908
2 -0.519439  0.281746
3 -0.555730  0.769023
4  1.965781  1.246435
object
  key1 key2
0    a  one
1    a  two
2    b  one
3    b  two
4    a  one
```

选取一列或列的子集

对于由DataFrame产生的GroupBy对象，如果用一个（单个字符串）或一组（字符串数组）列名对其进行索引，就能实现选取部分列进行聚合的目的。也就是说：

```
df.groupby('key1')['data1']
```

```
df.groupby('key1')[['data2']]
```

是以下代码的语法糖：

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

尤其对于大数据集，很可能只需要对部分列进行聚合。例如，在前面那个数据集中，如果只需计算data2列的平均值并以DataFrame形式得到结果，可以这样写：

```
In [31]: df.groupby(['key1', 'key2'])[['data2']].mean
Out[31]:
                data2
key1 key2
a    one   1.319920
     two   0.092908
b    one   0.281746
     two   0.769023
```

这种索引操作所返回的对象是一个已分组的DataFrame（如果传入的是列表或数组）或已分组的Series（如果传入的是标量形式的单个列名）：

```
In [32]: s_grouped = df.groupby(['key1', 'key2'])['da

In [33]: s_grouped
Out[33]: <pandas.core.groupby.SeriesGroupBy object at

In [34]: s_grouped.mean()
Out[34]:
key1  key2
a     one     1.319920
      two     0.092908
b     one     0.281746
      two     0.769023
```

```
Name: data2, dtype: float64
```

通过字典或Series进行分组

除数组以外，分组信息还可以其他形式存在。来看另一个示例DataFrame：

```
In [35]: people = pd.DataFrame(np.random.randn(5, 5),
   ....:                       columns=['a', 'b', 'c'
   ....:                       index=['Joe', 'Steve',

In [36]: people.iloc[2:3, [1, 2]] = np.nan # Add a fe

In [37]: people
Out[37]:
               a         b         c         d
Joe     1.007189 -1.296221  0.274992  0.228913  1.352
Steve   0.886429 -2.001637 -0.371843  1.669025 -0.438
Wes    -0.539741       NaN       NaN -1.021228 -0.577
Jim     0.124121  0.302614  0.523772  0.000940  1.343
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860
```

现在，假设已知列的分组关系，并希望根据分组计算列的和：

```
In [38]: mapping = {'a': 'red', 'b': 'red', 'c': 'blu
   ....:            'd': 'blue', 'e': 'red', 'f' : 'o
```

现在，你可以将这个字典传给groupby，来构造数组，但我们可以直接传递字典（我包含了键“f”来强调，存在未使用的分组键是可以的）：

```
In [39]: by_column = people.groupby(mapping, axis=1)

In [40]: by_column.sum()
Out[40]:
```

```
              blue       red
Joe      0.503905  1.063885
Steve    1.297183 -1.553778
Wes     -1.021228 -1.116829
Jim      0.524712  1.770545
Travis  -4.230992 -2.405455
```

Series也有同样的功能，它可以被看做一个固定大小的映射：

```
In [41]: map_series = pd.Series(mapping)

In [42]: map_series
Out[42]:
a       red
b       red
c      blue
d      blue
e       red
f    orange
dtype: object

In [43]: people.groupby(map_series, axis=1).count()
Out[43]:
        blue  red
Joe        2    3
Steve      2    3
Wes        1    2
Jim        2    3
Travis     2    3
```

通过函数进行分组

比起使用字典或Series，使用Python函数是一种更原生的方法定义分组映射。任何被当做分组键的函数都会在各个索引值上被调用一次，其返回值就会被用作分组名称。具体点说，以上一小节的示例DataFrame为例，其索引值为人的名字。你可以计算一个字符串长度的数组，更简单的方法是传入len函数：

```
In [44]: people.groupby(len).sum()
Out[44]:
          a         b         c         d         e
3  0.591569 -0.993608  0.798764 -0.791374  2.119639
5  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

将函数跟数组、列表、字典、Series混合使用也不是问题，因为任何东西在内部都会被转换为数组：

```
In [45]: key_list = ['one', 'one', 'one', 'two', 'two

In [46]: people.groupby([len, key_list]).min()
Out[46]:
              a         b         c         d
3 one -0.539741 -1.296221  0.274992 -1.021228 -0.5770
  two  0.124121  0.302614  0.523772  0.000940  1.3438
5 one  0.886429 -2.001637 -0.371843  1.669025 -0.4385
6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.8607
```

根据索引级别分组

层次化索引数据集最方便的地方就在于它能够根据轴索引的一个级别进行聚合：

```
In [47]: columns = pd.MultiIndex.from_arrays([['US',
   ....:                                      [1, 3, 5
   ....:                                      names=['

In [48]: hier_df = pd.DataFrame(np.random.randn(4, 5)

In [49]: hier_df
Out[49]:
cty          US                            JP
tenor         1         3         5         1
0      0.560145 -1.265934  0.119827 -1.063512  0.3328
```

```
1      -2.359419 -0.199543 -1.541996 -0.970736 -1.30703
2       0.286350  0.377984 -0.753887  0.331286  1.3497
3       0.069877  0.246674 -0.011862  1.004812  1.3271
```

要根据级别分组，使用level关键字传递级别序号或名字：

```
In [50]: hier_df.groupby(level='cty', axis=1).count()
Out[50]:
cty  JP  US
0     2   3
1     2   3
2     2   3
3     2   3
```

10.2 数据聚合

聚合指的是任何能够从数组产生标量值的数据转换过程。之前的例子已经用过一些，比如mean、count、min以及sum等。你可能想知道在GroupBy对象上调用mean()时究竟发生了什么。许多常见的聚合运算（如表10-1所示）都有进行优化。然而，除了这些方法，你还可以使用其它的。

函数名	说明
count	分组中非NA值的数量
sum	非NA值的和
mean	非NA值的平均值
median	非NA值的算术中位数
std、var	无偏（分母为n - 1）标准差和方差
min、max	非NA值的最小值和最大值
prod	非NA值的积
first、last	第一个和最后一个非NA值

表10-1 经过优化的groupby方法

你可以使用自己发明的聚合运算，还可以调用分组对象上已经定义好的任何方法。例如，quantile可以计算Series或DataFrame

列的样本分位数。

虽然quantile并没有明确地实现于GroupBy，但它是一个Series方法，所以这里是能用的。实际上，GroupBy会高效地对Series进行切片，然后对各片调用piece.quantile(0.9)，最后将这些结果组装成最终结果：

```
In [51]: df
Out[51]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one

In [52]: grouped = df.groupby('key1')

In [53]: grouped['data1'].quantile(0.9)
Out[53]:
key1
a    1.668413
b   -0.523068
Name: data1, dtype: float64
```

如果要使用你自己的聚合函数，只需将其传入aggregate或agg方法即可：

```
In [54]: def peak_to_peak(arr):
   ....:     return arr.max() - arr.min()
In [55]: grouped.agg(peak_to_peak)
Out[55]:
         data1     data2
key1
a     2.170488  1.300498
b     0.036292  0.487276
```

你可能注意到注意，有些方法（如describe）也是可以用在这里的，即使严格来讲，它们并非聚合运算：

```
In [56]: grouped.describe()
Out[56]:
     data1
     count      mean       std       min       25%
key1
a      3.0  0.746672  1.109736 -0.204708  0.137118  0
b      2.0 -0.537585  0.025662 -0.555730 -0.546657 -0
               data2
max count      mean       std       min       25%
key1
a     1.965781   3.0  0.910916  0.712217  0.092908  0
b    -0.519439   2.0  0.525384  0.344556  0.281746  0

           75%       max
key1
a     1.319920  1.393406
b     0.647203  0.769023
```

在后面的10.3节，我将详细说明这到底是怎么回事。

> 笔记：自定义聚合函数要比表10-1中那些经过优化的函数慢得多。这是因为在构造中间分组数据块时存在非常大的开销（函数调用、数据重排等）。

面向列的多函数应用

回到前面小费的例子。使用read_csv导入数据之后，我们添加了一个小费百分比的列tip_pct：

```
In [57]: tips = pd.read_csv('examples/tips.csv')

# Add tip percentage of total bill
In [58]: tips['tip_pct'] = tips['tip'] / tips['total_
```

```
In [59]: tips[:6]
Out[59]:
   total_bill   tip smoker  day    time  size   tip_p
0       16.99  1.01     No  Sun  Dinner     2  0.0594
1       10.34  1.66     No  Sun  Dinner     3  0.1605
2       21.01  3.50     No  Sun  Dinner     3  0.1665
3       23.68  3.31     No  Sun  Dinner     2  0.1397
4       24.59  3.61     No  Sun  Dinner     4  0.1468
5       25.29  4.71     No  Sun  Dinner     4  0.1862
```

你已经看到，对Series或DataFrame列的聚合运算其实就是使用aggregate（使用自定义函数）或调用诸如mean、std之类的方法。然而，你可能希望对不同的列使用不同的聚合函数，或一次应用多个函数。其实这也好办，我将通过一些示例来进行讲解。首先，我根据天和smoker对tips进行分组：

```
In [60]: grouped = tips.groupby(['day', 'smoker'])
```

注意，对于表10-1中的那些描述统计，可以将函数名以字符串的形式传入：

```
In [61]: grouped_pct = grouped['tip_pct']

In [62]: grouped_pct.agg('mean')
Out[62]:
day   smoker
Fri   No        0.151650
      Yes       0.174783
Sat   No        0.158048
      Yes       0.147906
Sun   No        0.160113
      Yes       0.187250
Thur  No        0.160298
      Yes       0.163863
Name: tip_pct, dtype: float64
```

如果传入一组函数或函数名，得到的DataFrame的列就会以相应的函数命名：

```
In [63]: grouped_pct.agg(['mean', 'std', peak_to_peak
Out[63]:
                  mean       std  peak_to_peak
day  smoker
Fri  No       0.151650  0.028123      0.067349
     Yes      0.174783  0.051293      0.159925
Sat  No       0.158048  0.039767      0.235193
     Yes      0.147906  0.061375      0.290095
Sun  No       0.160113  0.042347      0.193226
     Yes      0.187250  0.154134      0.644685
Thur No       0.160298  0.038774      0.193350
     Yes      0.163863  0.039389      0.151240
```

这里，我们传递了一组聚合函数进行聚合，独立对数据分组进行评估。

你并非一定要接受GroupBy自动给出的那些列名，特别是lambda函数，它们的名称是'<lambda>'，这样的辨识度就很低了（通过函数的**name**属性看看就知道了）。因此，如果传入的是一个由(name,function)元组组成的列表，则各元组的第一个元素就会被用作DataFrame的列名（可以将这种二元元组列表看做一个有序映射）：

```
In [64]: grouped_pct.agg([('foo', 'mean'), ('bar', np
Out[64]:
                   foo       bar
day  smoker
Fri  No       0.151650  0.028123
     Yes      0.174783  0.051293
Sat  No       0.158048  0.039767
     Yes      0.147906  0.061375
Sun  No       0.160113  0.042347
```

```
     Yes     0.187250  0.154134
Thur No      0.160298  0.038774
     Yes     0.163863  0.039389
```

对于DataFrame，你还有更多选择，你可以定义一组应用于全部列的一组函数，或不同的列应用不同的函数。假设我们想要对tip_pct和total_bill列计算三个统计信息：

```
In [65]: functions = ['count', 'mean', 'max']

In [66]: result = grouped['tip_pct', 'total_bill'].ag

In [67]: result
Out[67]:
            tip_pct                     total_bill
              count      mean       max      count
day  smoker
Fri  No           4  0.151650  0.187735          4  1
     Yes         15  0.174783  0.263480         15  1
Sat  No          45  0.158048  0.291990         45  1
     Yes         42  0.147906  0.325733         42  2
Sun  No          57  0.160113  0.252672         57  2
     Yes         19  0.187250  0.710345         19  2
Thur No          45  0.160298  0.266312         45  1
     Yes         17  0.163863  0.241255         17  1
```

如你所见，结果DataFrame拥有层次化的列，这相当于分别对各列进行聚合，然后用concat将结果组装到一起，使用列名用作keys参数：

```
In [68]: result['tip_pct']
Out[68]:
             count      mean       max
day  smoker
Fri  No          4  0.151650  0.187735
```

```
      Yes         15  0.174783  0.263480
Sat   No          45  0.158048  0.291990
      Yes         42  0.147906  0.325733
Sun   No          57  0.160113  0.252672
      Yes         19  0.187250  0.710345
Thur  No          45  0.160298  0.266312
      Yes         17  0.163863  0.241255
```

跟前面一样，这里也可以传入带有自定义名称的一组元组：

```
In [69]: ftuples = [('Durchschnitt', 'mean'),('Abweic

In [70]: grouped['tip_pct', 'total_bill'].agg(ftuples
Out[70]:
                  tip_pct              total_bill
             Durchschnitt Abweichung Durchschnitt  Abw
day  smoker
Fri  No          0.151650   0.000791    18.420000   25
     Yes         0.174783   0.002631    16.813333   82
Sat  No          0.158048   0.001581    19.661778   79
     Yes         0.147906   0.003767    21.276667  101
Sun  No          0.160113   0.001793    20.506667   66
     Yes         0.187250   0.023757    24.120000  109
Thur No          0.160298   0.001503    17.113111   59
     Yes         0.163863   0.001551    19.190588   69
```

现在，假设你想要对一个列或不同的列应用不同的函数。具体的办法是向agg传入一个从列名映射到函数的字典：

```
In [71]: grouped.agg({'tip' : np.max, 'size' : 'sum'}
Out[71]:
               tip  size
day  smoker
Fri  No       3.50     9
     Yes      4.73    31
Sat  No       9.00   115
```

```
     Yes      10.00   104
Sun  No        6.00   167
     Yes       6.50    49
Thur No        6.70   112
     Yes       5.00    40

In [72]: grouped.agg({'tip_pct' : ['min', 'max', 'mea
   ....:              'size' : 'sum'})
Out[72]:
              tip_pct                                  s
                  min       max      mean       std   
day  smoker
Fri  No      0.120385  0.187735  0.151650  0.028123
     Yes     0.103555  0.263480  0.174783  0.051293
Sat  No      0.056797  0.291990  0.158048  0.039767  
     Yes     0.035638  0.325733  0.147906  0.061375  
Sun  No      0.059447  0.252672  0.160113  0.042347  
     Yes     0.065660  0.710345  0.187250  0.154134
Thur No      0.072961  0.266312  0.160298  0.038774  
     Yes     0.090014  0.241255  0.163863  0.039389
```

只有将多个函数应用到至少一列时，DataFrame才会拥有层次化的列。

以“没有行索引”的形式返回聚合数据

到目前为止，所有示例中的聚合数据都有由唯一的分组键组成的索引（可能还是层次化的）。由于并不总是需要如此，所以你可以向groupby传入as_index=False以禁用该功能：

```
In [73]: tips.groupby(['day', 'smoker'], as_index=Fal
Out[73]:
    day smoker  total_bill       tip      size   tip_
0   Fri     No   18.420000  2.812500  2.250000  0.151
1   Fri    Yes   16.813333  2.714000  2.066667  0.174
2   Sat     No   19.661778  3.102889  2.555556  0.158
```

```
3   Sat    Yes   21.276667  2.875476  2.476190  0.147
4   Sun     No   20.506667  3.167895  2.929825  0.160
5   Sun    Yes   24.120000  3.516842  2.578947  0.187
6  Thur     No   17.113111  2.673778  2.488889  0.160
7  Thur    Yes   19.190588  3.030000  2.352941  0.163
```

当然，对结果调用reset_index也能得到这种形式的结果。使用as_index=False方法可以避免一些不必要的计算。

10.3 apply：一般性的“拆分－应用－合并”

最通用的GroupBy方法是apply，本节剩余部分将重点讲解它。如图10-2所示，apply会将待处理的对象拆分成多个片段，然后对各片段调用传入的函数，最后尝试将各片段组合到一起。

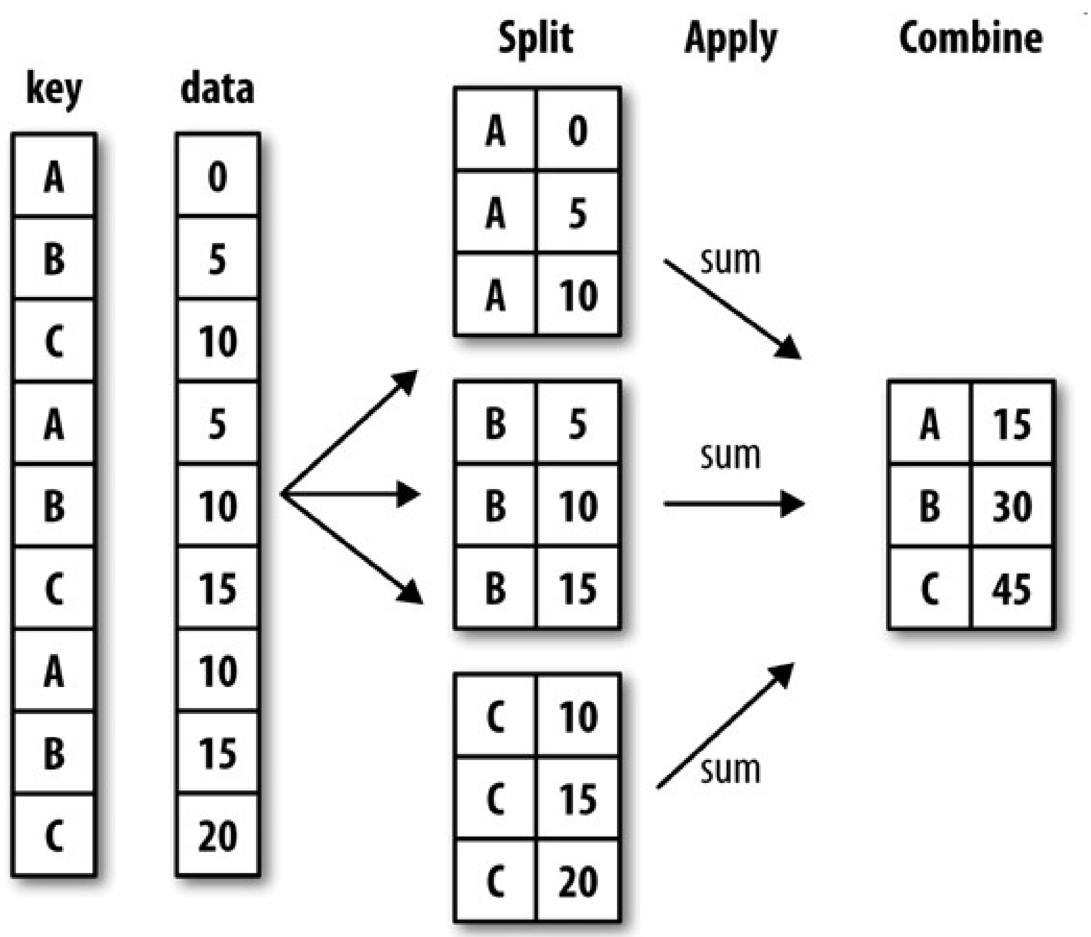


图10-2 分组聚合示例

回到之前那个小费数据集，假设你想要根据分组选出最高的5个tip_pct值。首先，编写一个选取指定列具有最大值的行的函数：

```
In [74]: def top(df, n=5, column='tip_pct'):
   ....:     return df.sort_values(by=column)[-n:]

In [75]: top(tips, n=6)
Out[75]:
     total_bill   tip smoker  day    time  size   tip_
109       14.31  4.00    Yes  Sat  Dinner     2  0.27
183       23.17  6.50    Yes  Sun  Dinner     4  0.28
232       11.61  3.39     No  Sat  Dinner     2  0.29
67         3.07  1.00    Yes  Sat  Dinner     1  0.32
178        9.60  4.00    Yes  Sun  Dinner     2  0.41
172        7.25  5.15    Yes  Sun  Dinner     2  0.71
```

现在，如果对smoker分组并用该函数调用apply，就会得到：

```
In [76]: tips.groupby('smoker').apply(top)
Out[76]:
            total_bill   tip smoker   day    time  si
smoker
No     88        24.71  5.85     No  Thur   Lunch
       185       20.69  5.00     No   Sun  Dinner
       51        10.29  2.60     No   Sun  Dinner
       149        7.51  2.00     No  Thur   Lunch
       232       11.61  3.39     No   Sat  Dinner
Yes    109       14.31  4.00    Yes   Sat  Dinner
       183       23.17  6.50    Yes   Sun  Dinner
       67         3.07  1.00    Yes   Sat  Dinner
       178        9.60  4.00    Yes   Sun  Dinner
       172        7.25  5.15    Yes   Sun  Dinner
```

这里发生了什么？top函数在DataFrame的各个片段上调用，然后结果由pandas.concat组装到一起，并以分组名称进行了标

记。于是，最终结果就有了一个层次化索引，其内层索引值来自原DataFrame。

如果传给apply的函数能够接受其他参数或关键字，则可以将这些内容放在函数名后面一并传入：

```
In [77]: tips.groupby(['smoker', 'day']).apply(top, n=
Out[77]:
                 total_bill    tip smoker   day    ti
smoker day
No     Fri  94        22.75   3.25     No   Fri  Dinn
       Sat  212       48.33   9.00     No   Sat  Dinn
       Sun  156       48.17   5.00     No   Sun  Dinn
       Thur 142       41.19   5.00     No  Thur   Lun
Yes    Fri  95        40.17   4.73    Yes   Fri  Dinn
       Sat  170       50.81  10.00    Yes   Sat  Dinn
       Sun  182       45.35   3.50    Yes   Sun  Dinn
       Thur 197       43.11   5.00    Yes  Thur   Lun
```

> 笔记：除这些基本用法之外，能否充分发挥apply的威力很大程度上取决于你的创造力。传入的那个函数能做什么全由你说了算，它只需返回一个pandas对象或标量值即可。本章后续部分的示例主要用于讲解如何利用groupby解决各种各样的问题。

可能你已经想起来了，之前我在GroupBy对象上调用过describe：

```
In [78]: result = tips.groupby('smoker')['tip_pct'].de

In [79]: result
Out[79]:
        count      mean       std       min       25%
smoker
No      151.0  0.159328  0.039910  0.056797  0.136906
Yes      93.0  0.163196  0.085119  0.035638  0.106771
```

```
               max
smoker

No       0.291990
Yes      0.710345

In [80]: result.unstack('smoker')
Out[80]:
       smoker
count  No        151.000000
       Yes        93.000000
mean   No          0.159328
       Yes         0.163196
std    No          0.039910
       Yes         0.085119
min    No          0.056797
       Yes         0.035638
25%    No          0.136906
       Yes         0.106771
50%    No          0.155625
       Yes         0.153846
75%    No          0.185014
       Yes         0.195059
max    No          0.291990
       Yes         0.710345
dtype: float64
```

在GroupBy中，当你调用诸如describe之类的方法时，实际上只是应用了下面两条代码的快捷方式而已：

```
f = lambda x: x.describe()
grouped.apply(f)
```

禁止分组键

从上面的例子中可以看出，分组键会跟原始对象的索引共同构成

结果对象中的层次化索引。将group_keys=False传入groupby即可禁止该效果：

```
In [81]: tips.groupby('smoker', group_keys=False).app
Out[81]:
     total_bill   tip smoker   day    time  size   ti
88        24.71  5.85     No  Thur   Lunch     2  0.2
185       20.69  5.00     No   Sun  Dinner     5  0.2
51        10.29  2.60     No   Sun  Dinner     2  0.2
149        7.51  2.00     No  Thur   Lunch     2  0.2
232       11.61  3.39     No   Sat  Dinner     2  0.2
109       14.31  4.00    Yes   Sat  Dinner     2  0.2
183       23.17  6.50    Yes   Sun  Dinner     4  0.2
67         3.07  1.00    Yes   Sat  Dinner     1  0.3
178        9.60  4.00    Yes   Sun  Dinner     2  0.4
172        7.25  5.15    Yes   Sun  Dinner     2  0.7
```

分位数和桶分析

我曾在第8章中讲过，pandas有一些能根据指定面元或样本分位数将数据拆分成多块的工具（比如cut和qcut）。将这些函数跟groupby结合起来，就能非常轻松地实现对数据集的桶（bucket）或分位数（quantile）分析了。以下面这个简单的随机数据集为例，我们利用cut将其装入长度相等的桶中：

```
In [82]: frame = pd.DataFrame({'data1': np.random.ran
   ....:                       'data2': np.random.ran

In [83]: quartiles = pd.cut(frame.data1, 4)

In [84]: quartiles[:10]
Out[84]:
0     (-1.23, 0.489]
1    (-2.956, -1.23]
2     (-1.23, 0.489]
3     (0.489, 2.208]
```

```
4     (-1.23, 0.489]
5     (0.489, 2.208]
6     (-1.23, 0.489]
7     (-1.23, 0.489]
8     (0.489, 2.208]
9     (0.489, 2.208]
Name: data1, dtype: category
Categories (4, interval[float64]): [(-2.956, -1.23] <
208] < (2.208, 3.928]]
```

由cut返回的Categorical对象可直接传递到groupby。因此，我们可以像下面这样对data2列做一些统计计算：

```
In [85]: def get_stats(group):
   ....:     return {'min': group.min(), 'max': group
   ....:             'count': group.count(), 'mean': 

In [86]: grouped = frame.data2.groupby(quartiles)

In [87]: grouped.apply(get_stats).unstack()
Out[87]:
                 count       max      mean       min
data1
(-2.956, -1.23]   95.0  1.670835 -0.039521 -3.399312
(-1.23, 0.489]   598.0  3.260383 -0.002051 -2.989741
(0.489, 2.208]   297.0  2.954439  0.081822 -3.745356
(2.208, 3.928]    10.0  1.765640  0.024750 -1.929776
```

这些都是长度相等的桶。要根据样本分位数得到大小相等的桶，使用qcut即可。传入labels=False即可只获取分位数的编号：

```
# Return quantile numbers
In [88]: grouping = pd.qcut(frame.data1, 10, labels=F

In [89]: grouped = frame.data2.groupby(grouping)
```

```
In [90]: grouped.apply(get_stats).unstack()
Out[90]:
       count       max      mean       min
data1
0      100.0  1.670835 -0.049902 -3.399312
1      100.0  2.628441  0.030989 -1.950098
2      100.0  2.527939 -0.067179 -2.925113
3      100.0  3.260383  0.065713 -2.315555
4      100.0  2.074345 -0.111653 -2.047939
5      100.0  2.184810  0.052130 -2.989741
6      100.0  2.458842 -0.021489 -2.223506
7      100.0  2.954439 -0.026459 -3.056990
8      100.0  2.735527  0.103406 -3.745356
9      100.0  2.377020  0.220122 -2.064111
```

我们会在第12章详细讲解pandas的Categorical类型。

示例：用特定于分组的值填充缺失值

对于缺失数据的清理工作，有时你会用dropna将其替换掉，而有时则可能会希望用一个固定值或由数据集本身所衍生出来的值去填充NA值。这时就得使用fillna这个工具了。在下面这个例子中，我用平均值去填充NA值：

```
In [91]: s = pd.Series(np.random.randn(6))

In [92]: s[::2] = np.nan

In [93]: s
Out[93]:
0         NaN
1   -0.125921
2         NaN
3   -0.884475
4         NaN
5    0.227290
```

```
dtype: float64

In [94]: s.fillna(s.mean())
Out[94]:
0   -0.261035
1   -0.125921
2   -0.261035
3   -0.884475
4   -0.261035
5    0.227290
dtype: float64
```

假设你需要对不同的分组填充不同的值。一种方法是将数据分组，并使用apply和一个能够对各数据块调用fillna的函数即可。下面是一些有关美国几个州的示例数据，这些州又被分为东部和西部：

```
In [95]: states = ['Ohio', 'New York', 'Vermont', 'Fl
   ....:           'Oregon', 'Nevada', 'California', 

In [96]: group_key = ['East'] * 4 + ['West'] * 4

In [97]: data = pd.Series(np.random.randn(8), index=s

In [98]: data
Out[98]:
Ohio          0.922264
New York     -2.153545
Vermont      -0.365757
Florida      -0.375842
Oregon        0.329939
Nevada        0.981994
California    1.105913
Idaho        -1.613716
dtype: float64
```

['East'] * 4产生了一个列表，包括了['East']中元素的四个拷贝。将这些列表串联起来。

将一些值设为缺失：

```
In [99]: data[['Vermont', 'Nevada', 'Idaho']] = np.na

In [100]: data
Out[100]:
Ohio          0.922264
New York     -2.153545
Vermont            NaN
Florida      -0.375842
Oregon        0.329939
Nevada             NaN
California    1.105913
Idaho              NaN
dtype: float64

In [101]: data.groupby(group_key).mean()
Out[101]:
East   -0.535707
West    0.717926
dtype: float64
```

我们可以用分组平均值去填充NA值：

```
In [102]: fill_mean = lambda g: g.fillna(g.mean())

In [103]: data.groupby(group_key).apply(fill_mean)
Out[103]:
Ohio          0.922264
New York     -2.153545
Vermont      -0.535707
Florida      -0.375842
Oregon        0.329939
```

```
Nevada        0.717926
California    1.105913
Idaho         0.717926
dtype: float64
```

外，也可以在代码中预定义各组的填充值。由于分组具有一个name属性，所以我们可以拿来用一下：

```
In [104]: fill_values = {'East': 0.5, 'West': -1}

In [105]: fill_func = lambda g: g.fillna(fill_values[

In [106]: data.groupby(group_key).apply(fill_func)
Out[106]:
Ohio          0.922264
New York     -2.153545
Vermont       0.500000
Florida      -0.375842
Oregon        0.329939
Nevada       -1.000000
California    1.105913
Idaho        -1.000000
dtype: float64
```

示例：随机采样和排列

假设你想要从一个大数据集中随机抽取（进行替换或不替换）样本以进行蒙特卡罗模拟（Monte Carlo simulation）或其他分析工作。“抽取”的方式有很多，这里使用的方法是对Series使用sample方法：

```
# Hearts, Spades, Clubs, Diamonds
suits = ['H', 'S', 'C', 'D']
card_val = (list(range(1, 11)) + [10] * 3) * 4
base_names = ['A'] + list(range(2, 11)) + ['J', 'K',
cards = []
```

```
for suit in ['H', 'S', 'C', 'D']:
    cards.extend(str(num) + suit for num in base_name

deck = pd.Series(card_val, index=cards)
```

现在我有了一个长度为52的Series，其索引包括牌名，值则是21点或其他游戏中用于计分的点数（为了简单起见，我当A的点数为1）：

```
In [108]: deck[:13]
Out[108]:
AH      1
2H      2
3H      3
4H      4
5H      5
6H      6
7H      7
8H      8
9H      9
10H    10
JH     10
KH     10
QH     10
dtype: int64
```

现在，根据我上面所讲的，从整副牌中抽出5张，代码如下：

```
In [109]: def draw(deck, n=5):
   .....:     return deck.sample(n)

In [110]: draw(deck)
Out[110]:
AD     1
8C     8
5H     5
```

```
KC    10
2C     2
dtype: int64
```

假设你想要从每种花色中随机抽取两张牌。由于花色是牌名的最后一个字符，所以我们可以据此进行分组，并使用apply：

```
In [111]: get_suit = lambda card: card[-1] # last let

In [112]: deck.groupby(get_suit).apply(draw, n=2)
Out[112]:
C  2C     2
   3C     3
D  KD    10
   8D     8
H  KH    10
   3H     3
S  2S     2
   4S     4
dtype: int64
```

或者，也可以这样写：

```
In [113]: deck.groupby(get_suit, group_keys=False).ap
Out[113]:
KC    10
JC    10
AD     1
5D     5
5H     5
6H     6
7S     7
KS    10
dtype: int64
```

示例：分组加权平均数和相关系数

根据groupby的“拆分－应用－合并”范式，可以进行DataFrame的列与列之间或两个Series之间的运算（比如分组加权平均）。以下面这个数据集为例，它含有分组键、值以及一些权重值：

```
In [114]: df = pd.DataFrame({'category': ['a', 'a', '
   .....:                                 'b', 'b', '
   .....:                    'data': np.random.randn(
   .....:                    'weights': np.random.ran

In [115]: df
Out[115]:
  category      data   weights
0        a  1.561587  0.957515
1        a  1.219984  0.347267
2        a -0.482239  0.581362
3        a  0.315667  0.217091
4        b -0.047852  0.894406
5        b -0.454145  0.918564
6        b -0.556774  0.277825
7        b  0.253321  0.955905
```

然后可以利用category计算分组加权平均数：

```
In [116]: grouped = df.groupby('category')

In [117]: get_wavg = lambda g: np.average(g['data'], 

In [118]: grouped.apply(get_wavg)
Out[118]:
category
a    0.811643
b   -0.122262
dtype: float64
```

另一个例子，考虑一个来自Yahoo!Finance的数据集，其中含有几只股票和标准普尔500指数（符号SPX）的收盘价：

```
In [119]: close_px = pd.read_csv('examples/stock_px_2
   .....:                         index_col=0)

In [120]: close_px.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 to 2011-10-14
Data columns (total 4 columns):
AAPL    2214 non-null float64
MSFT    2214 non-null float64
XOM     2214 non-null float64
SPX     2214 non-null float64
dtypes: float64(4)
memory usage: 86.5 KB

In [121]: close_px[-4:]
Out[121]:
              AAPL   MSFT    XOM      SPX
2011-10-11  400.29  27.00  76.27  1195.54
2011-10-12  402.19  26.96  77.16  1207.25
2011-10-13  408.43  27.18  76.37  1203.66
2011-10-14  422.00  27.27  78.11  1224.58
```

来做一个比较有趣的任务：计算一个由日收益率（通过百分数变化计算）与SPX之间的年度相关系数组成的DataFrame。下面是一个实现办法，我们先创建一个函数，用它计算每列和SPX列的成对相关系数：

```
In [122]: spx_corr = lambda x: x.corrwith(x['SPX'])
```

接下来，我们使用pct_change计算close_px的百分比变化：

```
In [123]: rets = close_px.pct_change().dropna()
```

最后，我们用年对百分比变化进行分组，可以用一个一行的函数，从每行的标签返回每个datetime标签的year属性：

```
In [124]: get_year = lambda x: x.year

In [125]: by_year = rets.groupby(get_year)

In [126]: by_year.apply(spx_corr)
Out[126]:
          AAPL      MSFT       XOM  SPX
2003  0.541124  0.745174  0.661265  1.0
2004  0.374283  0.588531  0.557742  1.0
2005  0.467540  0.562374  0.631010  1.0
2006  0.428267  0.406126  0.518514  1.0
2007  0.508118  0.658770  0.786264  1.0
2008  0.681434  0.804626  0.828303  1.0
2009  0.707103  0.654902  0.797921  1.0
2010  0.710105  0.730118  0.839057  1.0
2011  0.691931  0.800996  0.859975  1.0
```

当然，你还可以计算列与列之间的相关系数。这里，我们计算Apple和Microsoft的年相关系数：

```
In [127]: by_year.apply(lambda g: g['AAPL'].corr(g['MS
Out[127]:
2003    0.480868
2004    0.259024
2005    0.300093
2006    0.161735
2007    0.417738
2008    0.611901
2009    0.432738
2010    0.571946
2011    0.581987
dtype: float64
```

示例：组级别的线性回归

顺着上一个例子继续，你可以用groupby执行更为复杂的分组统计分析，只要函数返回的是pandas对象或标量值即可。例如，我可以定义下面这个regress函数（利用statsmodels计量经济学库）对各数据块执行普通最小二乘法（Ordinary Least Squares，OLS）回归：

```
import statsmodels.api as sm
def regress(data, yvar, xvars):
    Y = data[yvar]
    X = data[xvars]
    X['intercept'] = 1.
    result = sm.OLS(Y, X).fit()
    return result.params
```

现在，为了按年计算AAPL对SPX收益率的线性回归，执行：

```
In [129]: by_year.apply(regress, 'AAPL', ['SPX'])
Out[129]:
           SPX  intercept
2003  1.195406   0.000710
2004  1.363463   0.004201
2005  1.766415   0.003246
2006  1.645496   0.000080
2007  1.198761   0.003438
2008  0.968016  -0.001110
2009  0.879103   0.002954
2010  1.052608   0.001261
2011  0.806605   0.001514
```

10.4 透视表和交叉表

透视表（pivot table）是各种电子表格程序和其他数据分析软件中一种常见的数据汇总工具。它根据一个或多个键对数据进行聚合，并根据行和列上的分组键将数据分配到各个矩形区域中。在

Python和pandas中，可以通过本章所介绍的groupby功能以及（能够利用层次化索引的）重塑运算制作透视表。DataFrame有一个pivot_table方法，此外还有一个顶级的pandas.pivot_table函数。除能为groupby提供便利之外，pivot_table还可以添加分项小计，也叫做margins。

回到小费数据集，假设我想要根据day和smoker计算分组平均数（pivot_table的默认聚合类型），并将day和smoker放到行上：

```
In [130]: tips.pivot_table(index=['day', 'smoker'])
Out[130]:
                 size       tip   tip_pct  total_bill
day  smoker
Fri  No      2.250000  2.812500  0.151650   18.420000
     Yes     2.066667  2.714000  0.174783   16.813333
Sat  No      2.555556  3.102889  0.158048   19.661778
     Yes     2.476190  2.875476  0.147906   21.276667
Sun  No      2.929825  3.167895  0.160113   20.506667
     Yes     2.578947  3.516842  0.187250   24.120000
Thur No      2.488889  2.673778  0.160298   17.113111
     Yes     2.352941  3.030000  0.163863   19.190588
```

可以用groupby直接来做。现在，假设我们只想聚合tip_pct和size，而且想根据time进行分组。我将smoker放到列上，把day放到行上：

```
In [131]: tips.pivot_table(['tip_pct', 'size'], index=
   .....:                  columns='smoker')
Out[131]:
                 size             tip_pct
smoker             No       Yes        No       Yes
time   day
Dinner Fri   2.000000  2.222222  0.139622  0.165347
       Sat   2.555556  2.476190  0.158048  0.147906
       Sun   2.929825  2.578947  0.160113  0.187250
       Thur  2.000000       NaN  0.159744       NaN
```

```
Lunch  Fri    3.000000  1.833333  0.187735  0.188937
       Thur   2.500000  2.352941  0.160311  0.163863
```

还可以对这个表作进一步的处理，传入margins=True添加分项小计。这将会添加标签为All的行和列，其值对应于单个等级中所有数据的分组统计：

```
In [132]: tips.pivot_table(['tip_pct', 'size'], index
   .....:                  columns='smoker', margins=
Out[132]:
                  size                        tip_pct
smoker              No       Yes       All        No
time   day
Dinner Fri    2.000000  2.222222  2.166667  0.139622
       Sat    2.555556  2.476190  2.517241  0.158048
       Sun    2.929825  2.578947  2.842105  0.160113
       Thur   2.000000       NaN  2.000000  0.159744
Lunch  Fri    3.000000  1.833333  2.000000  0.187735
       Thur   2.500000  2.352941  2.459016  0.160311
All           2.668874  2.408602  2.569672  0.159328
```

这里，All值为平均数：不单独考虑烟民与非烟民（All列），不单独考虑行分组两个级别中的任何单项（All行）。

要使用其他的聚合函数，将其传给aggfunc即可。例如，使用count或len可以得到有关分组大小的交叉表（计数或频率）：

```
In [133]: tips.pivot_table('tip_pct', index=['time',
   .....:                  aggfunc=len, margins=True)
Out[133]:
day             Fri   Sat   Sun  Thur    All
time   smoker
Dinner No       3.0  45.0  57.0   1.0  106.0
       Yes      9.0  42.0  19.0   NaN   70.0
Lunch  No       1.0   NaN   NaN  44.0   45.0
       Yes      6.0   NaN   NaN  17.0   23.0
```

```
All           19.0  87.0  76.0  62.0  244.0
```

如果存在空的组合（也就是NA），你可能会希望设置一个fill_value：

```
In [134]: tips.pivot_table('tip_pct', index=['time', 
   .....:                  columns='day', aggfunc='me
Out[134]:
day                         Fri       Sat       Sun 
time   size smoker
Dinner 1    No       0.000000  0.137931  0.000000  0.0
            Yes      0.000000  0.325733  0.000000  0.0
       2    No       0.139622  0.162705  0.168859  0.1
            Yes      0.171297  0.148668  0.207893  0.0
       3    No       0.000000  0.154661  0.152663  0.0
            Yes      0.000000  0.144995  0.152660  0.0
       4    No       0.000000  0.150096  0.148143  0.0
            Yes      0.117750  0.124515  0.193370  0.0
       5    No       0.000000  0.000000  0.206928  0.0
Yes       0.000000  0.106572  0.065660  0.000000
...                         ...       ...       ...
Lunch  1    No       0.000000  0.000000  0.000000  0.1
            Yes      0.223776  0.000000  0.000000  0.0
       2    No       0.000000  0.000000  0.000000  0.1
            Yes      0.181969  0.000000  0.000000  0.1
       3    No       0.187735  0.000000  0.000000  0.0
            Yes      0.000000  0.000000  0.000000  0.2
       4    No       0.000000  0.000000  0.000000  0.1
            Yes      0.000000  0.000000  0.000000  0.1
       5    No       0.000000  0.000000  0.000000  0.1
       6    No       0.000000  0.000000  0.000000  0.1
[21 rows x 4 columns]
```

pivot_table的参数说明请参见表10-2。

函数名	说明
values	待聚合的列的名称。默认聚合所有数值列
index	用于分组的列名或其他分组键，出现在结果透视表的行
columns	用于分组的列名或其他分组键，出现在结果透视表的列
aggfunc	聚合函数或函数列表，默认为 mean。可以是任何对 groupby 有效的函数
fill_value	用于替换结果表中的缺失值
dropna	如果为 True，不添加条目都为 NA 的列
margins	添加行/列小计和总计，默认为 False

表10-2 pivot_table的选项

交叉表：crosstab

交叉表（cross-tabulation，简称crosstab）是一种用于计算分组频率的特殊透视表。看下面的例子：

```
In [138]: data
Out[138]:
   Sample Nationality    Handedness
0       1         USA  Right-handed
1       2       Japan   Left-handed
2       3         USA  Right-handed
3       4       Japan  Right-handed
4       5       Japan   Left-handed
5       6       Japan  Right-handed
6       7         USA  Right-handed
7       8         USA   Left-handed
8       9       Japan  Right-handed
9      10         USA  Right-handed
```

作为调查分析的一部分，我们可能想要根据国籍和用手习惯对这段数据进行统计汇总。虽然可以用pivot_table实现该功能，但是pandas.crosstab函数会更方便：

```
In [139]: pd.crosstab(data.Nationality, data.Handedne
Out[139]:
Handedness   Left-handed  Right-handed  All
Nationality
```

```
Japan                      2              3    5
USA                        1              4    5
All                        3              7   10
```

crosstab的前两个参数可以是数组或Series，或是数组列表。就像小费数据：

```
In [140]: pd.crosstab([tips.time, tips.day], tips.smo
Out[140]:
smoker        No  Yes  All
time   day
Dinner Fri     3    9   12
       Sat    45   42   87
       Sun    57   19   76
       Thur    1    0    1
Lunch  Fri     1    6    7
       Thur   44   17   61
All          151   93  244
```

10.5 总结

掌握pandas数据分组工具既有助于数据清理，也有助于建模或统计分析工作。在第14章，我们会看几个例子，对真实数据使用groupby。

在下一章，我们将关注时间序列数据。

第11章 时间序列

时间序列（time series）数据是一种重要的结构化数据形式，应用于多个领域，包括金融学、经济学、生态学、神经科学、物理学等。在多个时间点观察或测量到的任何事物都可以形成一段时间序列。很多时间序列是固定频率的，也就是说，数据点是根据某种规律定期出现的（比如每15秒、每5分钟、每月出现一次）。时间序列也可以是不定期的，没有固定的时间单位或单位之间的偏移量。时间序列数据的意义取决于具体的应用场景，主要有以下几种：

- 时间戳（timestamp），特定的时刻。
- 固定时期（period），如2007年1月或2010年全年。
- 时间间隔（interval），由起始和结束时间戳表示。时期（period）可以被看做间隔（interval）的特例。
- 实验或过程时间，每个时间点都是相对于特定起始时间的一个度量。例如，从放入烤箱时起，每秒钟饼干的直径。

本章主要讲解前3种时间序列。许多技术都可用于处理实验型时间序列，其索引可能是一个整数或浮点数（表示从实验开始算起已经过去的时间）。最简单也最常见的时间序列都是用时间戳进行索引的。

> 提示：pandas也支持基于timedeltas的指数，它可以有效代表实验或经过的时间。这本书不涉及timedelta指数，但你可以学习pandas的文档（http://pandas.pydata.org/）。

pandas提供了许多内置的时间序列处理工具和数据算法。因此，你可以高效处理非常大的时间序列，轻松地进行切片/切块、聚合、对定期/不定期的时间序列进行重采样等。有些工具特别适合金融和经济应用，你当然也可以用它们来分析服务器日志数据。

11.1 日期和时间数据类型及工具

Python标准库包含用于日期（date）和时间（time）数据的数据

类型，而且还有日历方面的功能。我们主要会用到datetime、time以及calendar模块。datetime.datetime（也可以简写为datetime）是用得最多的数据类型：

```
In [10]: from datetime import datetime

In [11]: now = datetime.now()

In [12]: now
Out[12]: datetime.datetime(2017, 9, 25, 14, 5, 52, 72

In [13]: now.year, now.month, now.day
Out[13]: (2017, 9, 25)
```

datetime以毫秒形式存储日期和时间。timedelta表示两个datetime对象之间的时间差：

```
In [14]: delta = datetime(2011, 1, 7) - datetime(2008

In [15]: delta
Out[15]: datetime.timedelta(926, 56700)

In [16]: delta.days
Out[16]: 926

In [17]: delta.seconds
Out[17]: 56700
```

可以给datetime对象加上（或减去）一个或多个timedelta，这样会产生一个新对象：

```
In [18]: from datetime import timedelta

In [19]: start = datetime(2011, 1, 7)
```

```
In [20]: start + timedelta(12)
Out[20]: datetime.datetime(2011, 1, 19, 0, 0)

In [21]: start - 2 * timedelta(12)
Out[21]: datetime.datetime(2010, 12, 14, 0, 0)
```

datetime模块中的数据类型参见表10-1。虽然本章主要讲的是pandas数据类型和高级时间序列处理，但你肯定会在Python的其他地方遇到有关datetime的数据类型。

表11-1 datetime模块中的数据类型

类型	说明
date	以公历形式存储日历日期（年、月、日）
time	将时间存储为时、分、秒、毫秒
datetime	存储日期和时间
timedelta	表示两个datetime值之间的差（日、秒、毫秒）

tzinfo 存储时区信息的基本类型

字符串和datetime的相互转换

利用str或strftime方法（传入一个格式化字符串），datetime对象和pandas的Timestamp对象（稍后就会介绍）可以被格式化为字符串：

```
In [22]: stamp = datetime(2011, 1, 3)

In [23]: str(stamp)
Out[23]: '2011-01-03 00:00:00'

In [24]: stamp.strftime('%Y-%m-%d')
Out[24]: '2011-01-03'
```

表11-2列出了全部的格式化编码。

表11-2 datetime格式定义（兼容ISO C89）

代码	说明
%Y	4位数的年
%y	2位数的年
%m	2位数的月[01, 12]
%d	2位数的日[01, 31]
%H	时（24小时制）[00, 23]
%I	时（12小时制）[01, 12]
%M	2位数的分[00, 59]
%S	秒[00, 61]（秒60和61用于闰秒）
%w	用整数表示的星期几[0（星期天），6]
%U	每年的第几周[00, 53]。星期天被认为是每周的第一天，每年第一个星期天之前的那几天被认为是“第0周”
%W	每年的第几周[00, 53]。星期一被认为是每周的第一天，每年第一个星期一之前的那几天被认为是“第0周”
%z	以+HHMM或-HHMM表示的UTC时区偏移量，如果时区为naive[译注3]，则返回空字符串
%F	%Y-%m-%d简写形式，例如2012-4-18[译注4]
%D	%m/%d/%y简写形式，例如04/18/12

datetime.strptime可以用这些格式化编码将字符串转换为日期：

```
In [25]: value = '2011-01-03'

In [26]: datetime.strptime(value, '%Y-%m-%d')
Out[26]: datetime.datetime(2011, 1, 3, 0, 0)

In [27]: datestrs = ['7/6/2011', '8/6/2011']

In [28]: [datetime.strptime(x, '%m/%d/%Y') for x in d
Out[28]:
[datetime.datetime(2011, 7, 6, 0, 0),
 datetime.datetime(2011, 8, 6, 0, 0)]
```

datetime.strptime是通过已知格式进行日期解析的最佳方式。但是每次都要编写格式定义是很麻烦的事情，尤其是对于一些常见的日期格式。这种情况下，你可以用dateutil这个第三方包中的

parser.parse方法（pandas中已经自动安装好了）：

```
In [29]: from dateutil.parser import parse

In [30]: parse('2011-01-03')
Out[30]: datetime.datetime(2011, 1, 3, 0, 0)
```

dateutil可以解析几乎所有人类能够理解的日期表示形式：

```
In [31]: parse('Jan 31, 1997 10:45 PM')
Out[31]: datetime.datetime(1997, 1, 31, 22, 45)
```

在国际通用的格式中，日出现在月的前面很普遍，传入dayfirst=True即可解决这个问题：

```
In [32]: parse('6/12/2011', dayfirst=True)
Out[32]: datetime.datetime(2011, 12, 6, 0, 0)
```

pandas通常是用于处理成组日期的，不管这些日期是DataFrame的轴索引还是列。to_datetime方法可以解析多种不同的日期表示形式。对标准日期格式（如ISO8601）的解析非常快：

```
In [33]: datestrs = ['2011-07-06 12:00:00', '2011-08-

In [34]: pd.to_datetime(datestrs)
Out[34]: DatetimeIndex(['2011-07-06 12:00:00', '2011-
etime64[ns]', freq=None)
```

它还可以处理缺失值（None、空字符串等）：

```
In [35]: idx = pd.to_datetime(datestrs + [None])

In [36]: idx
Out[36]: DatetimeIndex(['2011-07-06 12:00:00', '2011-
pe='datetime64[ns]', freq=None)
```

```
In [37]: idx[2]
Out[37]: NaT

In [38]: pd.isnull(idx)
Out[38]: array([False, False,  True], dtype=bool)
```

NaT（Not a Time）是pandas中时间戳数据的null值。

> 注意：dateutil.parser是一个实用但不完美的工具。比如说，它会把一些原本不是日期的字符串认作是日期（比如"42"会被解析为2042年的今天）。

datetime对象还有一些特定于当前环境（位于不同国家或使用不同语言的系统）的格式化选项。例如，德语或法语系统所用的月份简写就与英语系统所用的不同。表11-3进行了总结。

表11-3 特定于当前环境的日期格式

代码	说明
%a	星期几的简写
%A	星期几的全称
%b	月份的简写
%B	月份的全称
%c	完整的日期和时间，例如“Tue 01 May 2012 04:20:57 PM”
%p	不同环境中的AM或PM
%x	适合于当前环境的日期格式，例如，在美国，“May 1, 2012”会产生“05/01/2012”
%X	适合于当前环境的时间格式，例如“04:24:12 PM”

11.2 时间序列基础

pandas最基本的时间序列类型就是以时间戳（通常以Python字符串或datatime对象表示）为索引的Series：

```
In [39]: from datetime import datetime
```

```
In [40]: dates = [datetime(2011, 1, 2), datetime(2011
   ....:          datetime(2011, 1, 7), datetime(2011
   ....:          datetime(2011, 1, 10), datetime(201

In [41]: ts = pd.Series(np.random.randn(6), index=dat

In [42]: ts
Out[42]: 
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

这些datetime对象实际上是被放在一个DatetimeIndex中的：

```
In [43]: ts.index
Out[43]: 
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-0
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

跟其他Series一样，不同索引的时间序列之间的算术运算会自动按日期对齐：

```
In [44]: ts + ts[::2]
Out[44]: 
2011-01-02   -0.409415
2011-01-05         NaN
2011-01-07   -1.038877
2011-01-08         NaN
2011-01-10    3.931561
2011-01-12         NaN
dtype: float64
```

ts[::2] 是每隔两个取一个。

pandas用NumPy的datetime64数据类型以纳秒形式存储时间戳：

```
In [45]: ts.index.dtype
Out[45]: dtype('<M8[ns]')
```

DatetimeIndex中的各个标量值是pandas的Timestamp对象：

```
In [46]: stamp = ts.index[0]

In [47]: stamp
Out[47]: Timestamp('2011-01-02 00:00:00')
```

只要有需要，TimeStamp可以随时自动转换为datetime对象。此外，它还可以存储频率信息（如果有的话），且知道如何执行时区转换以及其他操作。稍后将对此进行详细讲解。

索引、选取、子集构造

当你根据标签索引选取数据时，时间序列和其它的pandas.Series很像：

```
In [48]: stamp = ts.index[2]

In [49]: ts[stamp]
Out[49]: -0.51943871505673811
```

还有一种更为方便的用法：传入一个可以被解释为日期的字符串：

```
In [50]: ts['1/10/2011']
Out[50]: 1.9657805725027142

In [51]: ts['20110110']
```

```
Out[51]: 1.9657805725027142
```

对于较长的时间序列，只需传入“年”或“年月”即可轻松选取数据的切片：

```
In [52]: longer_ts = pd.Series(np.random.randn(1000),
   ....:                       index=pd.date_range('1/

In [53]: longer_ts
Out[53]:
2000-01-01    0.092908
2000-01-02    0.281746
2000-01-03    0.769023
2000-01-04    1.246435
2000-01-05    1.007189
2000-01-06   -1.296221
2000-01-07    0.274992
2000-01-08    0.228913
2000-01-09    1.352917
2000-01-10    0.886429
                ...
2002-09-17   -0.139298
2002-09-18   -1.159926
2002-09-19    0.618965
2002-09-20    1.373890
2002-09-21   -0.983505
2002-09-22    0.930944
2002-09-23   -0.811676
2002-09-24   -1.830156
2002-09-25   -0.138730
2002-09-26    0.334088
Freq: D, Length: 1000, dtype: float64

In [54]: longer_ts['2001']
Out[54]:
2001-01-01    1.599534
2001-01-02    0.474071
```

```
2001-01-03    0.151326
2001-01-04   -0.542173
2001-01-05   -0.475496
2001-01-06    0.106403
2001-01-07   -1.308228
2001-01-08    2.173185
2001-01-09    0.564561
2001-01-10   -0.190481
                ...
2001-12-22    0.000369
2001-12-23    0.900885
2001-12-24   -0.454869
2001-12-25   -0.864547
2001-12-26    1.129120
2001-12-27    0.057874
2001-12-28   -0.433739
2001-12-29    0.092698
2001-12-30   -1.397820
2001-12-31    1.457823
Freq: D, Length: 365, dtype: float64
```

这里，字符串“2001”被解释成年，并根据它选取时间区间。指定月也同样奏效：

```
In [55]: longer_ts['2001-05']
Out[55]:
2001-05-01   -0.622547
2001-05-02    0.936289
2001-05-03    0.750018
2001-05-04   -0.056715
2001-05-05    2.300675
2001-05-06    0.569497
2001-05-07    1.489410
2001-05-08    1.264250
2001-05-09   -0.761837
2001-05-10   -0.331617
```

```
                ...
2001-05-22    0.503699
2001-05-23   -1.387874
2001-05-24    0.204851
2001-05-25    0.603705
2001-05-26    0.545680
2001-05-27    0.235477
2001-05-28    0.111835
2001-05-29   -1.251504
2001-05-30   -2.949343
2001-05-31    0.634634
Freq: D, Length: 31, dtype: float64
```

datetime对象也可以进行切片：

```
In [56]: ts[datetime(2011, 1, 7):]
Out[56]:
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

由于大部分时间序列数据都是按照时间先后排序的，因此你也可以用不存在于该时间序列中的时间戳对其进行切片（即范围查询）：

```
In [57]: ts
Out[57]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64
```

```
In [58]: ts['1/6/2011':'1/11/2011']
Out[58]:
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
dtype: float64
```

跟之前一样，你可以传入字符串日期、datetime或Timestamp。注意，这样切片所产生的是源时间序列的视图，跟NumPy数组的切片运算是一样的。

这意味着，没有数据被复制，对切片进行修改会反映到原始数据上。

此外，还有一个等价的实例方法也可以截取两个日期之间TimeSeries：

```
In [59]: ts.truncate(after='1/9/2011')
Out[59]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
dtype: float64
```

面这些操作对DataFrame也有效。例如，对DataFrame的行进行索引：

```
In [60]: dates = pd.date_range('1/1/2000', periods=10

In [61]: long_df = pd.DataFrame(np.random.randn(100, 4
   ....:                         index=dates,
   ....:                         columns=['Colorado',
   ....:                                  'New York',

In [62]: long_df.loc['5-2001']
Out[62]:
```

```
             Colorado     Texas  New York      Ohio
2001-05-02 -0.006045  0.490094 -0.277186 -0.707213
2001-05-09 -0.560107  2.735527  0.927335  1.513906
2001-05-16  0.538600  1.273768  0.667876 -0.969206
2001-05-23  1.676091 -0.817649  0.050188  1.951312
2001-05-30  3.260383  0.963301  1.201206 -1.852001
```

带有重复索引的时间序列

在某些应用场景中，可能会存在多个观测数据落在同一个时间点上的情况。下面就是一个例子：

```
In [63]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2
   ....:                            '1/2/2000', '1/3/2
In [64]: dup_ts = pd.Series(np.arange(5), index=dates

In [65]: dup_ts
Out[65]:
2000-01-01    0
2000-01-02    1
2000-01-02    2
2000-01-02    3
2000-01-03    4
dtype: int64
```

通过检查索引的is_unique属性，我们就可以知道它是不是唯一的：

```
In [66]: dup_ts.index.is_unique
Out[66]: False
```

对这个时间序列进行索引，要么产生标量值，要么产生切片，具体要看所选的时间点是否重复：

```
In [67]: dup_ts['1/3/2000']  # not duplicated
Out[67]: 4
```

```
In [68]: dup_ts['1/2/2000']  # duplicated
Out[68]:
2000-01-02    1
2000-01-02    2
2000-01-02    3
dtype: int64
```

假设你想要对具有非唯一时间戳的数据进行聚合。一个办法是使用groupby，并传入level=0：

```
In [69]: grouped = dup_ts.groupby(level=0)

In [70]: grouped.mean()
Out[70]:
2000-01-01    0
2000-01-02    2
2000-01-03    4
dtype: int64

In [71]: grouped.count()
Out[71]:
2000-01-01    1
2000-01-02    3
2000-01-03    1
dtype: int64
```

11.3 日期的范围、频率以及移动

pandas中的原生时间序列一般被认为是不规则的，也就是说，它们没有固定的频率。对于大部分应用程序而言，这是无所谓的。但是，它常常需要以某种相对固定的频率进行分析，比如每日、每月、每15分钟等（这样自然会在时间序列中引入缺失值）。幸运的是，pandas有一整套标准时间序列频率以及用于重采样、频率推断、生成固定频率日期范围的工具。例如，我们可以将之前那个时间序列转换为一个具有固定频率（每日）的时

间序列，只需调用resample即可：

```
In [72]: ts
Out[72]:
2011-01-02   -0.204708
2011-01-05    0.478943
2011-01-07   -0.519439
2011-01-08   -0.555730
2011-01-10    1.965781
2011-01-12    1.393406
dtype: float64

In [73]: resampler = ts.resample('D')
```

字符串“D”是每天的意思。

频率的转换（或重采样）是一个比较大的主题，稍后将专门用一节来进行讨论（11.6小节）。这里，我将告诉你如何使用基本的频率和它的倍数。

生成日期范围

虽然我之前用的时候没有明说，但你可能已经猜到pandas.date_range可用于根据指定的频率生成指定长度的DatetimeIndex：

```
In [74]: index = pd.date_range('2012-04-01', '2012-06

In [75]: index
Out[75]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-0
               '2012-04-05', '2012-04-06', '2012-04-0
               '2012-04-09', '2012-04-10', '2012-04-1
               '2012-04-13', '2012-04-14', '2012-04-1
               '2012-04-17', '2012-04-18', '2012-04-1
               '2012-04-21', '2012-04-22', '2012-04-2
               '2012-04-25', '2012-04-26', '2012-04-2
```

```
               '2012-04-29', '2012-04-30', '2012-05-0
               '2012-05-03', '2012-05-04', '2012-05-0
               '2012-05-07', '2012-05-08', '2012-05-0
               '2012-05-11', '2012-05-12', '2012-05-1
               '2012-05-15', '2012-05-16', '2012-05-1
               '2012-05-19', '2012-05-20', '2012-05-2
               '2012-05-23', '2012-05-24', '2012-05-2
               '2012-05-27', '2012-05-28', '2012-05-2
               '2012-05-31', '2012-06-01'],
              dtype='datetime64[ns]', freq='D')
```

默认情况下，date_range会产生按天计算的时间点。如果只传入起始或结束日期，那就还得传入一个表示一段时间的数字：

```
In [76]: pd.date_range(start='2012-04-01', periods=20
Out[76]:
DatetimeIndex(['2012-04-01', '2012-04-02', '2012-04-0
               '2012-04-05', '2012-04-06', '2012-04-0
               '2012-04-09', '2012-04-10', '2012-04-1
               '2012-04-13', '2012-04-14', '2012-04-1
               '2012-04-17', '2012-04-18', '2012-04-1
              dtype='datetime64[ns]', freq='D')

In [77]: pd.date_range(end='2012-06-01', periods=20)
Out[77]:
DatetimeIndex(['2012-05-13', '2012-05-14', '2012-05-1
               '2012-05-17', '2012-05-18', '2012-05-1
               '2012-05-21', '2012-05-22', '2012-05-2
               '2012-05-25', '2012-05-26', '2012-05-2
               '2012-05-29', '2012-05-30', '2012-05-3
              dtype='datetime64[ns]', freq='D')
```

起始和结束日期定义了日期索引的严格边界。例如，如果你想要生成一个由每月最后一个工作日组成的日期索引，可以传入"BM"频率（表示business end of month，表11-4是频率列

表），这样就只会包含时间间隔内（或刚好在边界上的）符合频率要求的日期：

```
In [78]: pd.date_range('2000-01-01', '2000-12-01', fr
Out[78]:
DatetimeIndex(['2000-01-31', '2000-02-29', '2000-03-3
               '2000-05-31', '2000-06-30', '2000-07-3
               '2000-09-29', '2000-10-31', '2000-11-3
              dtype='datetime64[ns]', freq='BM')
```

表11-4 基本的时间序列频率（不完整）

别名	偏移量类型	说明
D	Day	每日历日
B	BusinessDay	每工作日
H	Hour	每小时
T或min	Minute	每分
S	Second	每秒
L或ms	Milli	每毫秒（即每千分之一秒）
U	Micro	每微秒（即每百万分之一秒）
M	MonthEnd	每月最后一个日历日
BM	BusinessMonthEnd	每月最后一个工作日
MS	MonthBegin	每月第一个日历日
BMS	BusinessMonthBegin	每月第一个工作日
W-MON、W-TUE…	Week	从指定的星期几（MON、TUE、WED、THU、FRI、SAT、SUN）开始算起，每周
WOM-1MON、WOM-2MON…	WeekOfMonth	产生每月第一、第二、第三或第四周的星期几。例如，WOM-3FRI表示每月第3个星期五
Q-JAN、Q-FEB…	QuarterEnd	对于以指定月份（JAN、FEB、MAR、APR、MAY、JUN、JUL、AUG、SEP、OCT、NOV、DEC）结束的年度，每季度最后一月的最后一个日历日
BQ-JAN、BQ-FEB…	BusinessQuarterEnd	对于以指定月份结束的年度，每季度最后一月的最后一个工作日

QS-JAN、QS-FEB…	QuarterBegin	对于以指定月份结束的年度，每季度最后一月的第一个日历日
BQS-JAN、BQS-FEB…	BusinessQuarterBegin	对于以指定月份结束的年度，每季度最后一月的第一个工作日
A-JAN、A-FEB…	YearEnd	每年指定月份（JAN、FEB、MAR、APR、MAY、JUN、JUL、AUG、SEP、OCT、NOV、DEC）的最后一个日历日
BA-JAN、BA-FEB…	BusinessYearEnd	每年指定月份的最后一个工作日
AS-JAN、AS-FEB…	YearBegin	每年指定月份的第一个日历日
BAS-JAN、BAS-FEB…	BusinessYearBegin	每年指定月份的第一个工作日

date_range默认会保留起始和结束时间戳的时间信息（如果有的话）：

```
In [79]: pd.date_range('2012-05-02 12:56:31', periods
Out[79]:
DatetimeIndex(['2012-05-02 12:56:31', '2012-05-03 12:
               '2012-05-04 12:56:31', '2012-05-05 12:
               '2012-05-06 12:56:31'],
              dtype='datetime64[ns]', freq='D')
```

有时，虽然起始和结束日期带有时间信息，但你希望产生一组被规范化（normalize）到午夜的时间戳。normalize选项即可实现该功能：

```
In [80]: pd.date_range('2012-05-02 12:56:31', periods
Out[80]:
DatetimeIndex(['2012-05-02', '2012-05-03', '2012-05-0
               '2012-05-06'],
              dtype='datetime64[ns]', freq='D')
```

频率和日期偏移量

pandas中的频率是由一个基础频率（base frequency）和一个乘数组成的。基础频率通常以一个字符串别名表示，比如"M"表示

每月，"H"表示每小时。对于每个基础频率，都有一个被称为日期偏移量（date offset）的对象与之对应。例如，按小时计算的频率可以用Hour类表示：

```
In [81]: from pandas.tseries.offsets import Hour, Min

In [82]: hour = Hour()

In [83]: hour
Out[83]: <Hour>
```

传入一个整数即可定义偏移量的倍数：

```
In [84]: four_hours = Hour(4)

In [85]: four_hours
Out[85]: <4 * Hours>
```

一般来说，无需明确创建这样的对象，只需使用诸如"H"或"4H"这样的字符串别名即可。在基础频率前面放上一个整数即可创建倍数：

```
In [86]: pd.date_range('2000-01-01', '2000-01-03 23:5
Out[86]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 04:
               '2000-01-01 08:00:00', '2000-01-01 12:
               '2000-01-01 16:00:00', '2000-01-01 20:
               '2000-01-02 00:00:00', '2000-01-02 04:
               '2000-01-02 08:00:00', '2000-01-02 12:
               '2000-01-02 16:00:00', '2000-01-02 20:
               '2000-01-03 00:00:00', '2000-01-03 04:
               '2000-01-03 08:00:00', '2000-01-03 12:
               '2000-01-03 16:00:00', '2000-01-03 20:
              dtype='datetime64[ns]', freq='4H')
```

大部分偏移量对象都可通过加法进行连接：

```
In [87]: Hour(2) + Minute(30)
Out[87]: <150 * Minutes>
```

同理，你也可以传入频率字符串（如"2h30min"），这种字符串可以被高效地解析为等效的表达式：

```
In [88]: pd.date_range('2000-01-01', periods=10, freq
Out[88]:
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:
               '2000-01-01 03:00:00', '2000-01-01 04:
               '2000-01-01 06:00:00', '2000-01-01 07:
               '2000-01-01 09:00:00', '2000-01-01 10:
               '2000-01-01 12:00:00', '2000-01-01 13:
              dtype='datetime64[ns]', freq='90T')
```

有些频率所描述的时间点并不是均匀分隔的。例如，"M"（日历月末）和"BM"（每月最后一个工作日）就取决于每月的天数，对于后者，还要考虑月末是不是周末。由于没有更好的术语，我将这些称为锚点偏移量（anchored offset）。

表11-4列出了pandas中的频率代码和日期偏移量类。

> 笔记：用户可以根据实际需求自定义一些频率类以便提供pandas所没有的日期逻辑，但具体的细节超出了本书的范围。

表11-4 时间序列的基础频率

别名	偏移量类型	说明
D	Day	每日历日
B	BusinessDay	每工作日
H	Hour	每小时
T或min	Minute	每分
S	Second	每秒
L或ms	Milli	每毫秒（即每千分之一秒）
U	Micro	每微秒（即每百万分之一秒）
M	MonthEnd	每月最后一个日历日
BM	BusinessMonthEnd	每月最后一个工作日
MS	MonthBegin	每月第一个日历日
BMS	BusinessMonthBegin	每月第一个工作日
W-MON、W-TUE…	Week	从指定的星期几（MON、TUE、WED、THU、FRI、SAT、SUN）开始算起，每周
WOM-1MON、WOM-2MON…	WeekOfMonth	产生每月第一、第二、第三或第四周的星期几。例如，WOM-3FRI表示每月第3个星期五
Q-JAN、Q-FEB…	QuarterEnd	对于以指定月份（JAN、FEB、MAR、APR、MAY、JUN、JUL、AUG、SEP、OCT、NOV、DEC）结束的年度，每季度最后一月的最后一个日历日
BQ-JAN、BQ-FEB…	BusinessQuarterEnd	对于以指定月份结束的年度，每季度最后一月的最后一个工作日
QS-JAN、QS-FEB…	QuarterBegin	对于以指定月份结束的年度，每季度最后一月的第一个日历日
BQS-JAN、BQS-FEB…	BusinessQuarterBegin	对于以指定月份结束的年度，每季度最后一月的第一个工作日
A-JAN、A-FEB…	YearEnd	每年指定月份（JAN、FEB、MAR、APR、MAY、JUN、JUL、AUG、SEP、OCT、NOV、DEC）的最后一个日历日
BA-JAN、BA-FEB…	BusinessYearEnd	每年指定月份的最后一个工作日
AS-JAN、AS-FEB…	YearBegin	每年指定月份的第一个日历日
BAS-JAN、BAS-FEB…	BusinessYearBegin	每年指定月份的第一个工作日

WOM日期

WOM（Week Of Month）是一种非常实用的频率类，它以WOM开头。它使你能获得诸如“每月第3个星期五”之类的日期：

```
In [89]: rng = pd.date_range('2012-01-01', '2012-09-0

In [90]: list(rng)
Out[90]:
[Timestamp('2012-01-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-02-17 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-03-16 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-04-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-05-18 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-06-15 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-07-20 00:00:00', freq='WOM-3FRI'),
 Timestamp('2012-08-17 00:00:00', freq='WOM-3FRI')]
```

移动（超前和滞后）数据

移动（shifting）指的是沿着时间轴将数据前移或后移。Series和DataFrame都有一个shift方法用于执行单纯的前移或后移操作，保持索引不变：

```
In [91]: ts = pd.Series(np.random.randn(4),
   ....:                index=pd.date_range('1/1/2000

In [92]: ts
Out[92]:
2000-01-31   -0.066748
2000-02-29    0.838639
2000-03-31   -0.117388
2000-04-30   -0.517795
Freq: M, dtype: float64

In [93]: ts.shift(2)
Out[93]:
2000-01-31         NaN
```

```
2000-02-29          NaN
2000-03-31   -0.066748
2000-04-30    0.838639
Freq: M, dtype: float64

In [94]: ts.shift(-2)
Out[94]:
2000-01-31   -0.117388
2000-02-29   -0.517795
2000-03-31         NaN
2000-04-30         NaN
Freq: M, dtype: float64
```

当我们这样进行移动时，就会在时间序列的前面或后面产生缺失数据。

shift通常用于计算一个时间序列或多个时间序列（如DataFrame的列）中的百分比变化。可以这样表达：

```
ts / ts.shift(1) - 1
```

由于单纯的移位操作不会修改索引，所以部分数据会被丢弃。因此，如果频率已知，则可以将其传给shift以便实现对时间戳进行位移而不是对数据进行简单位移：

```
In [95]: ts.shift(2, freq='M')
Out[95]:
2000-03-31   -0.066748
2000-04-30    0.838639
2000-05-31   -0.117388
2000-06-30   -0.517795
Freq: M, dtype: float64
```

这里还可以使用其他频率，于是你就能非常灵活地对数据进行超前和滞后处理了：

```
In [96]: ts.shift(3, freq='D')
Out[96]:
2000-02-03   -0.066748
2000-03-03    0.838639
2000-04-03   -0.117388
2000-05-03   -0.517795
dtype: float64

In [97]: ts.shift(1, freq='90T')
Out[97]:
2000-01-31 01:30:00   -0.066748
2000-02-29 01:30:00    0.838639
2000-03-31 01:30:00   -0.117388
2000-04-30 01:30:00   -0.517795
Freq: M, dtype: float64
```

通过偏移量对日期进行位移

pandas的日期偏移量还可以用在datetime或Timestamp对象上：

```
In [98]: from pandas.tseries.offsets import Day, Mont

In [99]: now = datetime(2011, 11, 17)

In [100]: now + 3 * Day()
Out[100]: Timestamp('2011-11-20 00:00:00')
```

如果加的是锚点偏移量（比如MonthEnd），第一次增量会将原日期向前滚动到符合频率规则的下一个日期：

```
In [101]: now + MonthEnd()
Out[101]: Timestamp('2011-11-30 00:00:00')

In [102]: now + MonthEnd(2)
Out[102]: Timestamp('2011-12-31 00:00:00')
```

通过锚点偏移量的rollforward和rollback方法，可明确地将日期向前或向后“滚动”：

```
In [103]: offset = MonthEnd()

In [104]: offset.rollforward(now)
Out[104]: Timestamp('2011-11-30 00:00:00')

In [105]: offset.rollback(now)
Out[105]: Timestamp('2011-10-31 00:00:00')
```

日期偏移量还有一个巧妙的用法，即结合groupby使用这两个“滚动”方法：

```
In [106]: ts = pd.Series(np.random.randn(20),
   .....:                index=pd.date_range('1/15/20

In [107]: ts
Out[107]:
2000-01-15   -0.116696
2000-01-19    2.389645
2000-01-23   -0.932454
2000-01-27   -0.229331
2000-01-31   -1.140330
2000-02-04    0.439920
2000-02-08   -0.823758
2000-02-12   -0.520930
2000-02-16    0.350282
2000-02-20    0.204395
2000-02-24    0.133445
2000-02-28    0.327905
2000-03-03    0.072153
2000-03-07    0.131678
2000-03-11   -1.297459
2000-03-15    0.997747
2000-03-19    0.870955
2000-03-23   -0.991253
```

```
2000-03-27    0.151699
2000-03-31    1.266151
Freq: 4D, dtype: float64

In [108]: ts.groupby(offset.rollforward).mean()
Out[108]:
2000-01-31   -0.005833
2000-02-29    0.015894
2000-03-31    0.150209
dtype: float64
```

当然，更简单、更快速地实现该功能的办法是使用resample（11.6小节将对此进行详细介绍）：

```
In [109]: ts.resample('M').mean()
Out[109]:
2000-01-31   -0.005833
2000-02-29    0.015894
2000-03-31    0.150209
Freq: M, dtype: float64
```

11.4 时区处理

时间序列处理工作中最让人不爽的就是对时区的处理。许多人都选择以协调世界时（UTC，它是格林尼治标准时间（Greenwich Mean Time）的接替者，目前已经是国际标准了）来处理时间序列。时区是以UTC偏移量的形式表示的。例如，夏令时期间，纽约比UTC慢4小时，而在全年其他时间则比UTC慢5小时。

在Python中，时区信息来自第三方库pytz，它使Python可以使用Olson数据库（汇编了世界时区信息）。这对历史数据非常重要，这是因为由于各地政府的各种突发奇想，夏令时转变日期（甚至UTC偏移量）已经发生过多次改变了。就拿美国来说，DST转变时间自1900年以来就改变过多次！

有关pytz库的更多信息，请查阅其文档。就本书而言，由于

pandas包装了pytz的功能，因此你可以不用记忆其API，只要记得时区的名称即可。时区名可以在shell中看到，也可以通过文档查看：

```
In [110]: import pytz

In [111]: pytz.common_timezones[-5:]
Out[111]: ['US/Eastern', 'US/Hawaii', 'US/Mountain',
```

要从pytz中获取时区对象，使用pytz.timezone即可：

```
In [112]: tz = pytz.timezone('America/New_York')

In [113]: tz
Out[113]: <DstTzInfo 'America/New_York' LMT-1 day, 19
```

pandas中的方法既可以接受时区名也可以接受这些对象。

时区本地化和转换

默认情况下，pandas中的时间序列是单纯的（naive）时区。看看下面这个时间序列：

```
In [114]: rng = pd.date_range('3/9/2012 9:30', period

In [115]: ts = pd.Series(np.random.randn(len(rng)), i

In [116]: ts
Out[116]:
2012-03-09 09:30:00   -0.202469
2012-03-10 09:30:00    0.050718
2012-03-11 09:30:00    0.639869
2012-03-12 09:30:00    0.597594
2012-03-13 09:30:00   -0.797246
2012-03-14 09:30:00    0.472879
Freq: D, dtype: float64
```

其索引的tz字段为None：

```
In [117]: print(ts.index.tz)
None
```

可以用时区集生成日期范围：

```
In [118]: pd.date_range('3/9/2012 9:30', periods=10, 
Out[118]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-
               '2012-03-11 09:30:00+00:00', '2012-03-
               '2012-03-13 09:30:00+00:00', '2012-03-
               '2012-03-15 09:30:00+00:00', '2012-03-
               '2012-03-17 09:30:00+00:00', '2012-03-
              dtype='datetime64[ns, UTC]', freq='D')
```

从单纯到本地化的转换是通过tz_localize方法处理的：

```
In [119]: ts
Out[119]:
2012-03-09 09:30:00   -0.202469
2012-03-10 09:30:00    0.050718
2012-03-11 09:30:00    0.639869
2012-03-12 09:30:00    0.597594
2012-03-13 09:30:00   -0.797246
2012-03-14 09:30:00    0.472879
Freq: D, dtype: float64

In [120]: ts_utc = ts.tz_localize('UTC')

In [121]: ts_utc
Out[121]:
2012-03-09 09:30:00+00:00   -0.202469
2012-03-10 09:30:00+00:00    0.050718
2012-03-11 09:30:00+00:00    0.639869
```

```
2012-03-12 09:30:00+00:00    0.597594
2012-03-13 09:30:00+00:00   -0.797246
2012-03-14 09:30:00+00:00    0.472879
Freq: D, dtype: float64

In [122]: ts_utc.index
Out[122]:
DatetimeIndex(['2012-03-09 09:30:00+00:00', '2012-03-
               '2012-03-11 09:30:00+00:00', '2012-03-
               '2012-03-13 09:30:00+00:00', '2012-03-
              dtype='datetime64[ns, UTC]', freq='D')
```

一旦时间序列被本地化到某个特定时区，就可以用tz_convert将其转换到别的时区了：

```
In [123]: ts_utc.tz_convert('America/New_York')
Out[123]:
2012-03-09 04:30:00-05:00   -0.202469
2012-03-10 04:30:00-05:00    0.050718
2012-03-11 05:30:00-04:00    0.639869
2012-03-12 05:30:00-04:00    0.597594
2012-03-13 05:30:00-04:00   -0.797246
2012-03-14 05:30:00-04:00    0.472879
Freq: D, dtype: float64
```

对于上面这种时间序列（它跨越了美国东部时区的夏令时转变期），我们可以将其本地化到EST，然后转换为UTC或柏林时间：

```
In [124]: ts_eastern = ts.tz_localize('America/New_Yo

In [125]: ts_eastern.tz_convert('UTC')
Out[125]:
2012-03-09 14:30:00+00:00   -0.202469
2012-03-10 14:30:00+00:00    0.050718
2012-03-11 13:30:00+00:00    0.639869
```

```
2012-03-12 13:30:00+00:00    0.597594
2012-03-13 13:30:00+00:00   -0.797246
2012-03-14 13:30:00+00:00    0.472879
Freq: D, dtype: float64

In [126]: ts_eastern.tz_convert('Europe/Berlin')
Out[126]:
2012-03-09 15:30:00+01:00   -0.202469
2012-03-10 15:30:00+01:00    0.050718
2012-03-11 14:30:00+01:00    0.639869
2012-03-12 14:30:00+01:00    0.597594
2012-03-13 14:30:00+01:00   -0.797246
2012-03-14 14:30:00+01:00    0.472879
Freq: D, dtype: float64
```

tz_localize和tz_convert也是DatetimeIndex的实例方法：

```
In [127]: ts.index.tz_localize('Asia/Shanghai')
Out[127]:
DatetimeIndex(['2012-03-09 09:30:00+08:00', '2012-03-
               '2012-03-11 09:30:00+08:00', '2012-03-
               '2012-03-13 09:30:00+08:00', '2012-03-
              dtype='datetime64[ns, Asia/Shanghai]',
```

> 注意：对单纯时间戳的本地化操作还会检查夏令时转变期附近容易混淆或不存在的时间。

操作时区意识型Timestamp对象

跟时间序列和日期范围差不多，独立的Timestamp对象也能被从单纯型（naive）本地化为时区意识型（time zone-aware），并从一个时区转换到另一个时区：

```
In [128]: stamp = pd.Timestamp('2011-03-12 04:00')

In [129]: stamp_utc = stamp.tz_localize('utc')
```

```
In [130]: stamp_utc.tz_convert('America/New_York')
Out[130]: Timestamp('2011-03-11 23:00:00-0500', tz='Ar
```

在创建Timestamp时，还可以传入一个时区信息：

```
In [131]: stamp_moscow = pd.Timestamp('2011-03-12 04:(

In [132]: stamp_moscow
Out[132]: Timestamp('2011-03-12 04:00:00+0300', tz='E(
```

时区意识型Timestamp对象在内部保存了一个UTC时间戳值（自UNIX纪元（1970年1月1日）算起的纳秒数）。这个UTC值在时区转换过程中是不会发生变化的：

```
In [133]: stamp_utc.value
Out[133]: 1299902400000000000

In [134]: stamp_utc.tz_convert('America/New_York').va
Out[134]: 1299902400000000000
```

当使用pandas的DateOffset对象执行时间算术运算时，运算过程会自动关注是否存在夏令时转变期。这里，我们创建了在DST转变之前的时间戳。首先，来看夏令时转变前的30分钟：

```
In [135]: from pandas.tseries.offsets import Hour

In [136]: stamp = pd.Timestamp('2012-03-12 01:30', tz=

In [137]: stamp
Out[137]: Timestamp('2012-03-12 01:30:00-0400', tz='U

In [138]: stamp + Hour()
Out[138]: Timestamp('2012-03-12 02:30:00-0400', tz='U
```

然后，夏令时转变前90分钟：

```
In [139]: stamp = pd.Timestamp('2012-11-04 00:30', tz=

In [140]: stamp
Out[140]: Timestamp('2012-11-04 00:30:00-0400', tz='US

In [141]: stamp + 2 * Hour()
Out[141]: Timestamp('2012-11-04 01:30:00-0500', tz='US
```

不同时区之间的运算

如果两个时间序列的时区不同，在将它们合并到一起时，最终结果就会是UTC。由于时间戳其实是以UTC存储的，所以这是一个很简单的运算，并不需要发生任何转换：

```
In [142]: rng = pd.date_range('3/7/2012 9:30', period

In [143]: ts = pd.Series(np.random.randn(len(rng)), in

In [144]: ts
Out[144]:
2012-03-07 09:30:00    0.522356
2012-03-08 09:30:00   -0.546348
2012-03-09 09:30:00   -0.733537
2012-03-12 09:30:00    1.302736
2012-03-13 09:30:00    0.022199
2012-03-14 09:30:00    0.364287
2012-03-15 09:30:00   -0.922839
2012-03-16 09:30:00    0.312656
2012-03-19 09:30:00   -1.128497
2012-03-20 09:30:00   -0.333488
Freq: B, dtype: float64

In [145]: ts1 = ts[:7].tz_localize('Europe/London')
```

```
In [146]: ts2 = ts1[2:].tz_convert('Europe/Moscow')

In [147]: result = ts1 + ts2

In [148]: result.index
Out[148]:
DatetimeIndex(['2012-03-07 09:30:00+00:00', '2012-03-(
               '2012-03-09 09:30:00+00:00', '2012-03-:
               '2012-03-13 09:30:00+00:00', '2012-03-:
               '2012-03-15 09:30:00+00:00'],
              dtype='datetime64[ns, UTC]', freq='B')
```

11.5 时期及其算术运算

时期（period）表示的是时间区间，比如数日、数月、数季、数年等。Period类所表示的就是这种数据类型，其构造函数需要用到一个字符串或整数，以及表11-4中的频率：

```
In [149]: p = pd.Period(2007, freq='A-DEC')

In [150]: p
Out[150]: Period('2007', 'A-DEC')
```

这里，这个Period对象表示的是从2007年1月1日到2007年12月31日之间的整段时间。只需对Period对象加上或减去一个整数即可达到根据其频率进行位移的效果：

```
In [151]: p + 5
Out[151]: Period('2012', 'A-DEC')

In [152]: p - 2
Out[152]: Period('2005', 'A-DEC')
```

如果两个Period对象拥有相同的频率，则它们的差就是它们之间的单位数量：

```
In [153]: pd.Period('2014', freq='A-DEC') - p
Out[153]: 7
```

period_range函数可用于创建规则的时期范围：

```
In [154]: rng = pd.period_range('2000-01-01', '2000-0

In [155]: rng
Out[155]: PeriodIndex(['2000-01', '2000-02', '2000-03
00-06'], dtype='period[M]', freq='M')
```

PeriodIndex类保存了一组Period，它可以在任何pandas数据结构中被用作轴索引：

```
In [156]: pd.Series(np.random.randn(6), index=rng)
Out[156]:
2000-01   -0.514551
2000-02   -0.559782
2000-03   -0.783408
2000-04   -1.797685
2000-05   -0.172670
2000-06    0.680215
Freq: M, dtype: float64
```

如果你有一个字符串数组，你也可以使用PeriodIndex类：

```
In [157]: values = ['2001Q3', '2002Q2', '2003Q1']

In [158]: index = pd.PeriodIndex(values, freq='Q-DEC'

In [159]: index
Out[159]: PeriodIndex(['2001Q3', '2002Q2', '2003Q1'],
='Q-DEC')
```

时期的频率转换

Period和PeriodIndex对象都可以通过其asfreq方法被转换成别的频率。假设我们有一个年度时期，希望将其转换为当年年初或年末的一个月度时期。该任务非常简单：

```
In [160]: p = pd.Period('2007', freq='A-DEC')

In [161]: p
Out[161]: Period('2007', 'A-DEC')

In [162]: p.asfreq('M', how='start')
Out[162]: Period('2007-01', 'M')

In [163]: p.asfreq('M', how='end')
Out[163]: Period('2007-12', 'M')
```

你可以将Period('2007','A-DEC')看做一个被划分为多个月度时期的时间段中的游标。图11-1对此进行了说明。对于一个不以12月结束的财政年度，月度子时期的归属情况就不一样了：

```
In [164]: p = pd.Period('2007', freq='A-JUN')

In [165]: p
Out[165]: Period('2007', 'A-JUN')

In [166]: p.asfreq('M', 'start')
Out[166]: Period('2006-07', 'M')

In [167]: p.asfreq('M', 'end')
Out[167]: Period('2007-06', 'M')
```

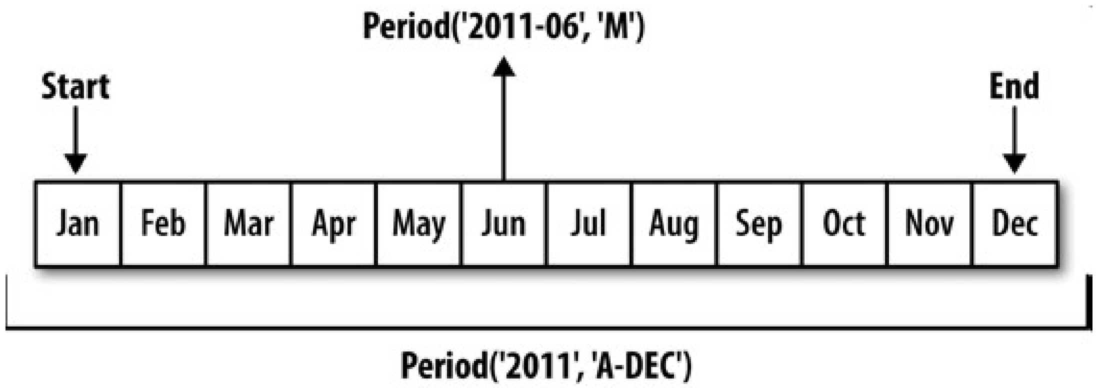


图11-1 Period频率转换示例

在将高频率转换为低频率时，超时期（superperiod）是由子时期（subperiod）所属的位置决定的。例如，在A-JUN频率中，月份“2007年8月”实际上是属于周期“2008年”的：

```
In [168]: p = pd.Period('Aug-2007', 'M')

In [169]: p.asfreq('A-JUN')
Out[169]: Period('2008', 'A-JUN')
```

完整的PeriodIndex或TimeSeries的频率转换方式也是如此：

```
In [170]: rng = pd.period_range('2006', '2009', freq=

In [171]: ts = pd.Series(np.random.randn(len(rng)), i

In [172]: ts
Out[172]:
2006    1.607578
2007    0.200381
2008   -0.834068
2009   -0.302988
Freq: A-DEC, dtype: float64

In [173]: ts.asfreq('M', how='start')
Out[173]:
2006-01    1.607578
2007-01    0.200381
```

```
2008-01   -0.834068
2009-01   -0.302988
Freq: M, dtype: float64
```

这里，根据年度时期的第一个月，每年的时期被取代为每月的时期。如果我们想要每年的最后一个工作日，我们可以使用“B”频率，并指明想要该时期的末尾：

```
In [174]: ts.asfreq('B', how='end')

Out[174]:
2006-12-29    1.607578
2007-12-31    0.200381
2008-12-31   -0.834068
2009-12-31   -0.302988
Freq: B, dtype: float64
```

按季度计算的时期频率

季度型数据在会计、金融等领域中很常见。许多季度型数据都会涉及“财年末”的概念，通常是一年12个月中某月的最后一个日历日或工作日。就这一点来说，时期"2012Q4"根据财年末的不同会有不同的含义。pandas支持12种可能的季度型频率，即Q-JAN到Q-DEC：

```
In [175]: p = pd.Period('2012Q4', freq='Q-JAN')

In [176]: p
Out[176]: Period('2012Q4', 'Q-JAN')
```

在以1月结束的财年中，2012Q4是从11月到1月（将其转换为日型频率就明白了）。图11-2对此进行了说明：

```
In [177]: p.asfreq('D', 'start')
Out[177]: Period('2011-11-01', 'D')
```

```
In [178]: p.asfreq('D', 'end')
Out[178]: Period('2012-01-31', 'D')
```



图11.2 不同季度型频率之间的转换

因此，Period之间的算术运算会非常简单。例如，要获取该季度倒数第二个工作日下午4点的时间戳，你可以这样：

```
In [179]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T',

In [180]: p4pm
Out[180]: Period('2012-01-30 16:00', 'T')

In [181]: p4pm.to_timestamp()
Out[181]: Timestamp('2012-01-30 16:00:00')
```

period_range可用于生成季度型范围。季度型范围的算术运算也跟上面是一样的：

```
In [182]: rng = pd.period_range('2011Q3', '2012Q4', f

In [183]: ts = pd.Series(np.arange(len(rng)), index=r

In [184]: ts
Out[184]:
2011Q3    0
2011Q4    1
2012Q1    2
2012Q2    3
```

```
2012Q3    4
2012Q4    5
Freq: Q-JAN, dtype: int64

In [185]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq

In [186]: ts.index = new_rng.to_timestamp()

In [187]: ts
Out[187]:
2010-10-28 16:00:00    0
2011-01-28 16:00:00    1
2011-04-28 16:00:00    2
2011-07-28 16:00:00    3
2011-10-28 16:00:00    4
2012-01-30 16:00:00    5
dtype: int64
```

将Timestamp转换为Period（及其反向过程）

通过使用to_period方法，可以将由时间戳索引的Series和DataFrame对象转换为以时期索引：

```
In [188]: rng = pd.date_range('2000-01-01', periods=3

In [189]: ts = pd.Series(np.random.randn(3), index=rn

In [190]: ts
Out[190]:
2000-01-31    1.663261
2000-02-29   -0.996206
2000-03-31    1.521760
Freq: M, dtype: float64

In [191]: pts = ts.to_period()
```

```
In [192]: pts
Out[192]:
2000-01    1.663261
2000-02   -0.996206
2000-03    1.521760
Freq: M, dtype: float64
```

由于时期指的是非重叠时间区间，因此对于给定的频率，一个时间戳只能属于一个时期。新PeriodIndex的频率默认是从时间戳推断而来的，你也可以指定任何别的频率。结果中允许存在重复时期：

```
In [193]: rng = pd.date_range('1/29/2000', periods=6,

In [194]: ts2 = pd.Series(np.random.randn(6), index=r

In [195]: ts2
Out[195]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
2000-02-03   -0.167933
Freq: D, dtype: float64

In [196]: ts2.to_period('M')
Out[196]:
2000-01    0.244175
2000-01    0.423331
2000-01   -0.654040
2000-02    2.089154
2000-02   -0.060220
2000-02   -0.167933
Freq: M, dtype: float64
```

要转换回时间戳，使用to_timestamp即可：

```
In [197]: pts = ts2.to_period()

In [198]: pts
Out[198]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
2000-02-03   -0.167933
Freq: D, dtype: float64

In [199]: pts.to_timestamp(how='end')
Out[199]:
2000-01-29    0.244175
2000-01-30    0.423331
2000-01-31   -0.654040
2000-02-01    2.089154
2000-02-02   -0.060220
2000-02-03   -0.167933
Freq: D, dtype: float64
```

通过数组创建PeriodIndex

固定频率的数据集通常会将时间信息分开存放在多个列中。例如，在下面这个宏观经济数据集中，年度和季度就分别存放在不同的列中：

```
In [200]: data = pd.read_csv('examples/macrodata.csv'

In [201]: data.head(5)
Out[201]:
     year  quarter   realgdp  realcons  realinv  real
0  1959.0      1.0  2710.349    1707.4  286.898   470
1  1959.0      2.0  2778.801    1733.7  310.859   481
```

```
2  1959.0      3.0  2775.488    1751.8  289.226   491
3  1959.0      4.0  2785.204    1753.7  299.356   484
4  1960.0      1.0  2847.699    1770.5  331.722   462
      m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82    5.8  177.146  0.00     0.00
1  141.7      3.08    5.1  177.830  2.34     0.74
2  140.5      3.82    5.3  178.657  2.74     1.09
3  140.0      4.33    5.6  179.386  0.27     4.06
4  139.6      3.50    5.2  180.007  2.31     1.19

In [202]: data.year
Out[202]:
0      1959.0
1      1959.0
2      1959.0
3      1959.0
4      1960.0
5      1960.0
6      1960.0
7      1960.0
8      1961.0
9      1961.0
        ...
193    2007.0
194    2007.0
195    2007.0
196    2008.0
197    2008.0
198    2008.0
199    2008.0
200    2009.0
201    2009.0
202    2009.0
Name: year, Length: 203, dtype: float64

In [203]: data.quarter
Out[203]:
```

```
0      1.0
1      2.0
2      3.0
3      4.0
4      1.0
5      2.0
6      3.0
7      4.0
8      1.0
9      2.0
      ...
193    2.0
194    3.0
195    4.0
196    1.0
197    2.0
198    3.0
199    4.0
200    1.0
201    2.0
202    3.0
Name: quarter, Length: 203, dtype: float64
```

通过通过将这些数组以及一个频率传入PeriodIndex，就可以将它们合并成DataFrame的一个索引：

```
In [204]: index = pd.PeriodIndex(year=data.year, quar
   .....:                        freq='Q-DEC')

In [205]: index
Out[205]:
PeriodIndex(['1959Q1', '1959Q2', '1959Q3', '1959Q4',
             '1960Q3', '1960Q4', '1961Q1', '1961Q2',
             ...
             '2007Q2', '2007Q3', '2007Q4', '2008Q1',
             '2008Q4', '2009Q1', '2009Q2', '2009Q3'],
```

```
             dtype='period[Q-DEC]', length=203, freq='(

In [206]: data.index = index

In [207]: data.infl
Out[207]:
1959Q1    0.00
1959Q2    2.34
1959Q3    2.74
1959Q4    0.27
1960Q1    2.31
1960Q2    0.14
1960Q3    2.70
1960Q4    1.21
1961Q1   -0.40
1961Q2    1.47
          ...
2007Q2    2.75
2007Q3    3.45
2007Q4    6.38
2008Q1    2.82
2008Q2    8.53
2008Q3   -3.16
2008Q4   -8.79
2009Q1    0.94
2009Q2    3.37
2009Q3    3.56
Freq: Q-DEC, Name: infl, Length: 203, dtype: float64
```

11.6 重采样及频率转换

重采样（resampling）指的是将时间序列从一个频率转换到另一个频率的处理过程。将高频率数据聚合到低频率称为降采样（downsampling），而将低频率数据转换到高频率则称为升采样（upsampling）。并不是所有的重采样都能被划分到这两个大类中。例如，将W-WED（每周三）转换为W-FRI既不是降采样

也不是升采样。

pandas对象都带有一个resample方法，它是各种频率转换工作的主力函数。resample有一个类似于groupby的API，调用resample可以分组数据，然后会调用一个聚合函数：

```
In [208]: rng = pd.date_range('2000-01-01', periods=10

In [209]: ts = pd.Series(np.random.randn(len(rng)), i

In [210]: ts
Out[210]:
2000-01-01    0.631634
2000-01-02   -1.594313
2000-01-03   -1.519937
2000-01-04    1.108752
2000-01-05    1.255853
2000-01-06   -0.024330
2000-01-07   -2.047939
2000-01-08   -0.272657
2000-01-09   -1.692615
2000-01-10    1.423830
                 ...
2000-03-31   -0.007852
2000-04-01   -1.638806
2000-04-02    1.401227
2000-04-03    1.758539
2000-04-04    0.628932
2000-04-05   -0.423776
2000-04-06    0.789740
2000-04-07    0.937568
2000-04-08   -2.253294
2000-04-09   -1.772919
Freq: D, Length: 100, dtype: float64

In [211]: ts.resample('M').mean()
Out[211]:
```

```
2000-01-31   -0.165893
2000-02-29    0.078606
2000-03-31    0.223811
2000-04-30   -0.063643
Freq: M, dtype: float64

In [212]: ts.resample('M', kind='period').mean()
Out[212]:
2000-01   -0.165893
2000-02    0.078606
2000-03    0.223811
2000-04   -0.063643
Freq: M, dtype: float64
```

resample是一个灵活高效的方法，可用于处理非常大的时间序列。我将通过一系列的示例说明其用法。表11-5总结它的一些选项。

表11-5 resample方法的参数

参数	说明
freq	表示重采样频率的字符串或 DeteOffset，例如'M'、'5min'或 Second(15)
axis	重采样的轴，默认为 axis=0
fill_method	升采样如何插值，比如'ffill'或'bfill'。默认不插值
closed	在降采样中，各时间段的哪一端是闭合（即包含）的，right 或 left。默认是 right
label	在降采样中，如何设置聚合值得标签，right 或 left（面元的右边界或左边界）。例如，9:30 到 9:35 之间的这 5 分钟会被标记为 9:30 或 9:35.默认为 right。
loffset	面元标签的时间校正值，比如'-1s'/Second(-1)用于将聚合标签调早 1 秒
limit	在前向或后向填充时，允许填充的最大时期数
kind	聚合到时期或时间戳，默认居合道时间序列的索引类型
convention	

降采样

将数据聚合到规律的低频率是一件非常普通的时间序列处理任务。待聚合的数据不必拥有固定的频率，期望的频率会自动定义聚合的面元边界，这些面元用于将时间序列拆分为多个片段。例如，要转换到月度频率（'M'或'BM'），数据需要被划分到多个单月时间段中。各时间段都是半开放的。一个数据点只能属于一个时间段，所有时间段的并集必须能组成整个时间帧。在用resample对数据进行降采样时，需要考虑两样东西：

- 各区间哪边是闭合的。
- 如何标记各个聚合面元，用区间的开头还是末尾。

为了说明，我们来看一些“1分钟”数据：

```
In [213]: rng = pd.date_range('2000-01-01', periods=1

In [214]: ts = pd.Series(np.arange(12), index=rng)

In [215]: ts
Out[215]:
2000-01-01 00:00:00     0
2000-01-01 00:01:00     1
2000-01-01 00:02:00     2
2000-01-01 00:03:00     3
2000-01-01 00:04:00     4
2000-01-01 00:05:00     5
2000-01-01 00:06:00     6
2000-01-01 00:07:00     7
2000-01-01 00:08:00     8
2000-01-01 00:09:00     9
2000-01-01 00:10:00    10
2000-01-01 00:11:00    11
Freq: T, dtype: int64
```

假设你想要通过求和的方式将这些数据聚合到“5分钟”块中：

```
In [216]: ts.resample('5min', closed='right').sum()
```

```
Out[216]:
1999-12-31 23:55:00     0
2000-01-01 00:00:00    15
2000-01-01 00:05:00    40
2000-01-01 00:10:00    11
Freq: 5T, dtype: int64
```

传入的频率将会以“5分钟”的增量定义面元边界。默认情况下，面元的右边界是包含的，因此00:00到00:05的区间中是包含00:05的。传入closed='left'会让区间以左边界闭合：

```
In [217]: ts.resample('5min', closed='right').sum()
Out[217]:
1999-12-31 23:55:00     0
2000-01-01 00:00:00    15
2000-01-01 00:05:00    40
2000-01-01 00:10:00    11
Freq: 5T, dtype: int64
```

如你所见，最终的时间序列是以各面元右边界的时间戳进行标记的。传入label='right'即可用面元的邮编界对其进行标记：

```
In [218]: ts.resample('5min', closed='right', label='
Out[218]:
2000-01-01 00:00:00     0
2000-01-01 00:05:00    15
2000-01-01 00:10:00    40
2000-01-01 00:15:00    11
Freq: 5T, dtype: int64
```

图11-3说明了“1分钟”数据被转换为“5分钟”数据的处理过程。

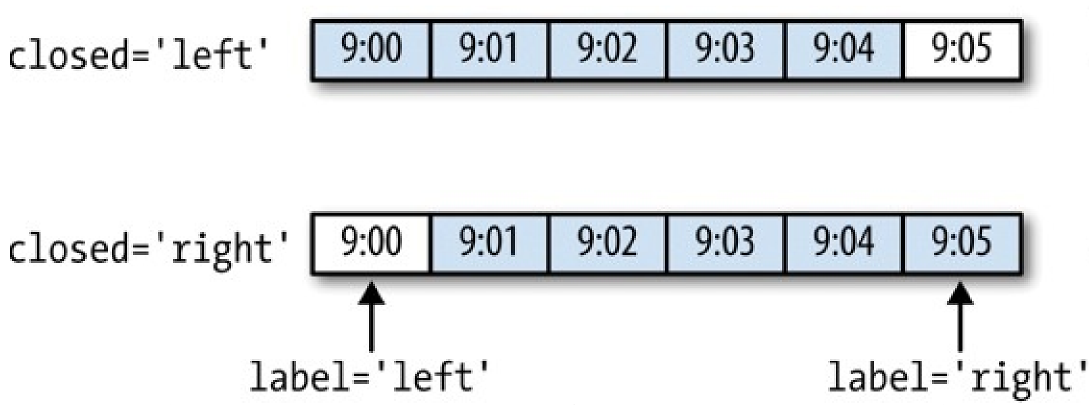


图11-3 各种closed、label约定的“5分钟”重采样演示

最后，你可能希望对结果索引做一些位移，比如从右边界减去一秒以便更容易明白该时间戳到底表示的是哪个区间。只需通过loffset设置一个字符串或日期偏移量即可实现这个目的：

```
In [219]: ts.resample('5min', closed='right',
   .....:              label='right', loffset='-1s').s
Out[219]:
1999-12-31 23:59:59     0
2000-01-01 00:04:59    15
In [219]: ts.resample('5min', closed='right',
   .....:              label='right', loffset='-1s').s
Out[219]:
1999-12-31 23:59:59     0
2000-01-01 00:04:59    15
```

此外，也可以通过调用结果对象的shift方法来实现该目的，这样就不需要设置loffset了。

OHLC重采样

金融领域中有一种无所不在的时间序列聚合方式，即计算各面元的四个值：第一个值（open，开盘）、最后一个值（close，收盘）、最大值（high，最高）以及最小值（low，最低）。传入how='ohlc'即可得到一个含有这四种聚合值的DataFrame。整个过程很高效，只需一次扫描即可计算出结果：

```
In [220]: ts.resample('5min').ohlc()
Out[220]:
                     open  high  low  close
2000-01-01 00:00:00     0     4    0      4
2000-01-01 00:05:00     5     9    5      9
2000-01-01 00:10:00    10    11   10     11
```

升采样和插值

在将数据从低频率转换到高频率时，就不需要聚合了。我们来看一个带有一些周型数据的DataFrame：

```
In [221]: frame = pd.DataFrame(np.random.randn(2, 4),
   .....:                      index=pd.date_range('1/
   .....:                                          fr
   .....:                      columns=['Colorado', '

In [222]: frame
Out[222]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

当你对这个数据进行聚合，每组只有一个值，这样就会引入缺失值。我们使用asfreq方法转换成高频，不经过聚合：

```
In [223]: df_daily = frame.resample('D').asfreq()

In [224]: df_daily
Out[224]:
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06       NaN       NaN       NaN       NaN
2000-01-07       NaN       NaN       NaN       NaN
2000-01-08       NaN       NaN       NaN       NaN
2000-01-09       NaN       NaN       NaN       NaN
```

```
2000-01-10       NaN       NaN       NaN       NaN
2000-01-11       NaN       NaN       NaN       NaN
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

假设你想要用前面的周型值填充“非星期三”。resampling的填充和插值方式跟fillna和reindex的一样：

```
In [225]: frame.resample('D').ffill()
Out[225]: 
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-07 -0.896431  0.677263  0.036503  0.087102
2000-01-08 -0.896431  0.677263  0.036503  0.087102
2000-01-09 -0.896431  0.677263  0.036503  0.087102
2000-01-10 -0.896431  0.677263  0.036503  0.087102
2000-01-11 -0.896431  0.677263  0.036503  0.087102
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

同样，这里也可以只填充指定的时期数（目的是限制前面的观测值的持续使用距离）：

```
In [226]: frame.resample('D').ffill(limit=2)
Out[226]: 
            Colorado     Texas  New York      Ohio
2000-01-05 -0.896431  0.677263  0.036503  0.087102
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-07 -0.896431  0.677263  0.036503  0.087102
2000-01-08       NaN       NaN       NaN       NaN
2000-01-09       NaN       NaN       NaN       NaN
2000-01-10       NaN       NaN       NaN       NaN
2000-01-11       NaN       NaN       NaN       NaN
2000-01-12 -0.046662  0.927238  0.482284 -0.867130
```

注意，新的日期索引完全没必要跟旧的重叠：

```
In [227]: frame.resample('W-THU').ffill()
```

```
Out[227]:
            Colorado     Texas  New York      Ohio
2000-01-06 -0.896431  0.677263  0.036503  0.087102
2000-01-13 -0.046662  0.927238  0.482284 -0.867130
```

通过时期进行重采样

对那些使用时期索引的数据进行重采样与时间戳很像：

```
In [228]: frame = pd.DataFrame(np.random.randn(24, 4)
   .....:                      index=pd.period_range(
   .....:                      
   .....:                      columns=['Colorado', '

In [229]: frame[:5]
Out[229]:
         Colorado     Texas  New York      Ohio
2000-01  0.493841 -0.155434  1.397286  1.507055
2000-02 -1.179442  0.443171  1.395676 -0.529658
2000-03  0.787358  0.248845  0.743239  1.267746
2000-04  1.302395 -0.272154 -0.051532 -0.467740
2000-05 -1.040816  0.426419  0.312945 -1.115689

In [230]: annual_frame = frame.resample('A-DEC').mean

In [231]: annual_frame
Out[231]:
      Colorado     Texas  New York      Ohio
2000  0.556703  0.016631  0.111873 -0.027445
2001  0.046303  0.163344  0.251503 -0.157276
```

升采样要稍微麻烦一些，因为你必须决定在新频率中各区间的哪端用于放置原来的值，就像asfreq方法那样。convention参数默认为'end'，可设置为'start'：

```
# Q-DEC: Quarterly, year ending in December
```

```
In [232]: annual_frame.resample('Q-DEC').ffill()
Out[232]:
        Colorado     Texas  New York      Ohio
2000Q1  0.556703  0.016631  0.111873 -0.027445
2000Q2  0.556703  0.016631  0.111873 -0.027445
2000Q3  0.556703  0.016631  0.111873 -0.027445
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.046303  0.163344  0.251503 -0.157276
2001Q2  0.046303  0.163344  0.251503 -0.157276
2001Q3  0.046303  0.163344  0.251503 -0.157276
2001Q4  0.046303  0.163344  0.251503 -0.157276

In [233]: annual_frame.resample('Q-DEC', convention='
Out[233]:
        Colorado     Texas  New York      Ohio
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.556703  0.016631  0.111873 -0.027445
2001Q2  0.556703  0.016631  0.111873 -0.027445
2001Q3  0.556703  0.016631  0.111873 -0.027445
2001Q4  0.046303  0.163344  0.251503 -0.157276
```

由于时期指的是时间区间，所以升采样和降采样的规则就比较严格：

- 在降采样中，目标频率必须是源频率的子时期（subperiod）。
- 在升采样中，目标频率必须是源频率的超时期（superperiod）。

如果不满足这些条件，就会引发异常。这主要影响的是按季、年、周计算的频率。例如，由Q-MAR定义的时间区间只能升采样为A-MAR、A-JUN、A-SEP、A-DEC等：

```
In [234]: annual_frame.resample('Q-MAR').ffill()
Out[234]:
        Colorado     Texas  New York      Ohio
```

```
2000Q4  0.556703  0.016631  0.111873 -0.027445
2001Q1  0.556703  0.016631  0.111873 -0.027445
2001Q2  0.556703  0.016631  0.111873 -0.027445
2001Q3  0.556703  0.016631  0.111873 -0.027445
2001Q4  0.046303  0.163344  0.251503 -0.157276
2002Q1  0.046303  0.163344  0.251503 -0.157276
2002Q2  0.046303  0.163344  0.251503 -0.157276
2002Q3  0.046303  0.163344  0.251503 -0.157276
```

11.7 移动窗口函数

在移动窗口（可以带有指数衰减权数）上计算的各种统计函数也是一类常见于时间序列的数组变换。这样可以圆滑噪音数据或断裂数据。我将它们称为移动窗口函数（moving window function），其中还包括那些窗口不定长的函数（如指数加权移动平均）。跟其他统计函数一样，移动窗口函数也会自动排除缺失值。

开始之前，我们加载一些时间序列数据，将其重采样为工作日频率：

```
In [235]: close_px_all = pd.read_csv('examples/stock_
   .....:                            parse_dates=True

In [236]: close_px = close_px_all[['AAPL', 'MSFT', 'X

In [237]: close_px = close_px.resample('B').ffill()
```

现在引入rolling运算符，它与resample和groupby很像。可以在TimeSeries或DataFrame以及一个window（表示期数，见图11-4）上调用它：

```
In [238]: close_px.AAPL.plot()
Out[238]: <matplotlib.axes._subplots.AxesSubplot at 0

In [239]: close_px.AAPL.rolling(250).mean().plot()
```

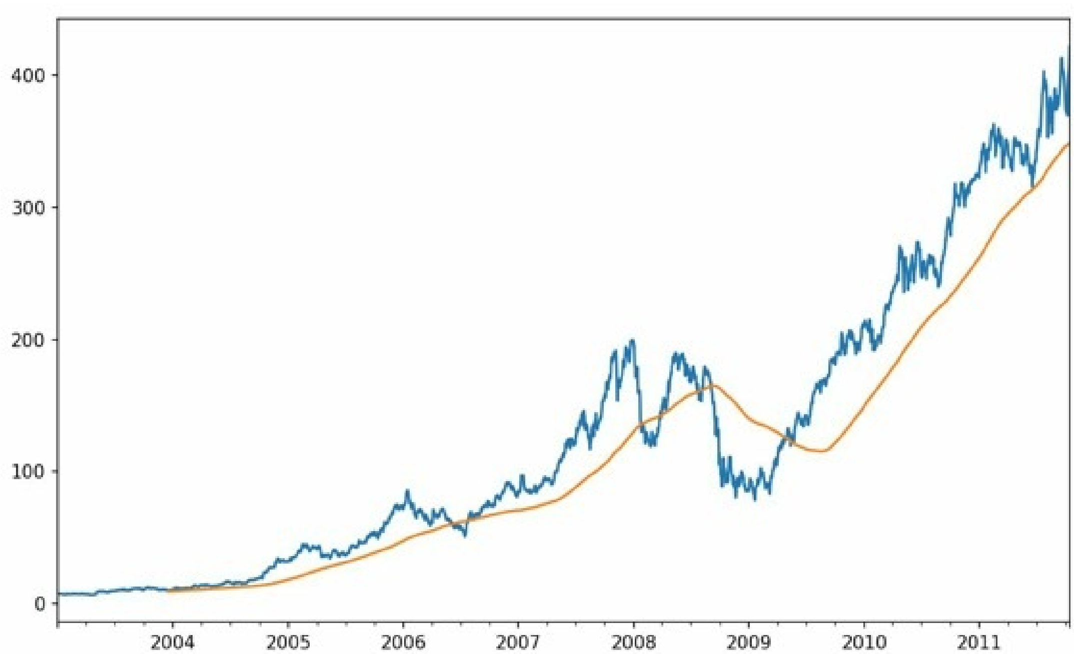


图11-4 苹果公司股价的250日均线

表达式rolling(250)与groupby很像，但不是对其进行分组、创建一个按照250天分组的滑动窗口对象。然后，我们就得到了苹果公司股价的250天的移动窗口。

默认情况下，诸如rolling_mean这样的函数需要指定数量的非NA观测值。可以修改该行为以解决缺失数据的问题。其实，在时间序列开始处尚不足窗口期的那些数据就是个特例（见图11-5）：

```
In [241]: appl_std250 = close_px.AAPL.rolling(250, mi

In [242]: appl_std250[5:12]
Out[242]:
2003-01-09         NaN
2003-01-10         NaN
2003-01-13         NaN
2003-01-14         NaN
2003-01-15    0.077496
2003-01-16    0.074760
2003-01-17    0.112368
```

```
Freq: B, Name: AAPL, dtype: float64

In [243]: appl_std250.plot()
```

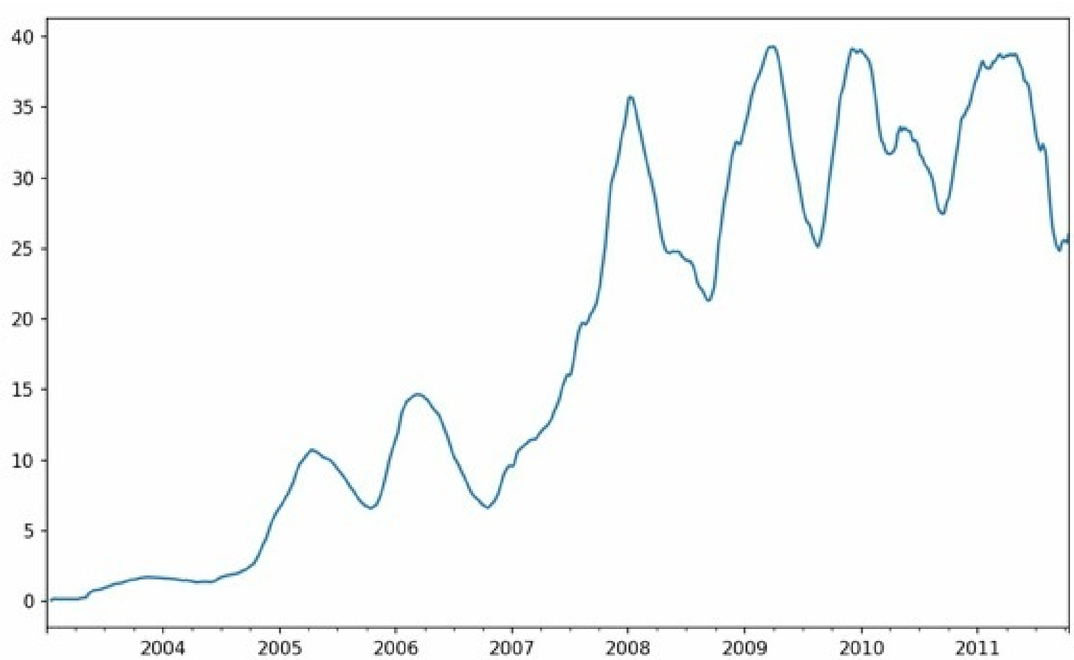


图11-5 苹果公司250日每日回报标准差

要计算扩展窗口平均（expanding window mean），可以使用expanding而不是rolling。“扩展”意味着，从时间序列的起始处开始窗口，增加窗口直到它超过所有的序列。apple_std250时间序列的扩展窗口平均如下所示：

```
In [244]: expanding_mean = appl_std250.expanding().mea
```

对DataFrame调用rolling_mean（以及与之类似的函数）会将转换应用到所有的列上（见图11-6）：

```
In [246]: close_px.rolling(60).mean().plot(logy=True)
```

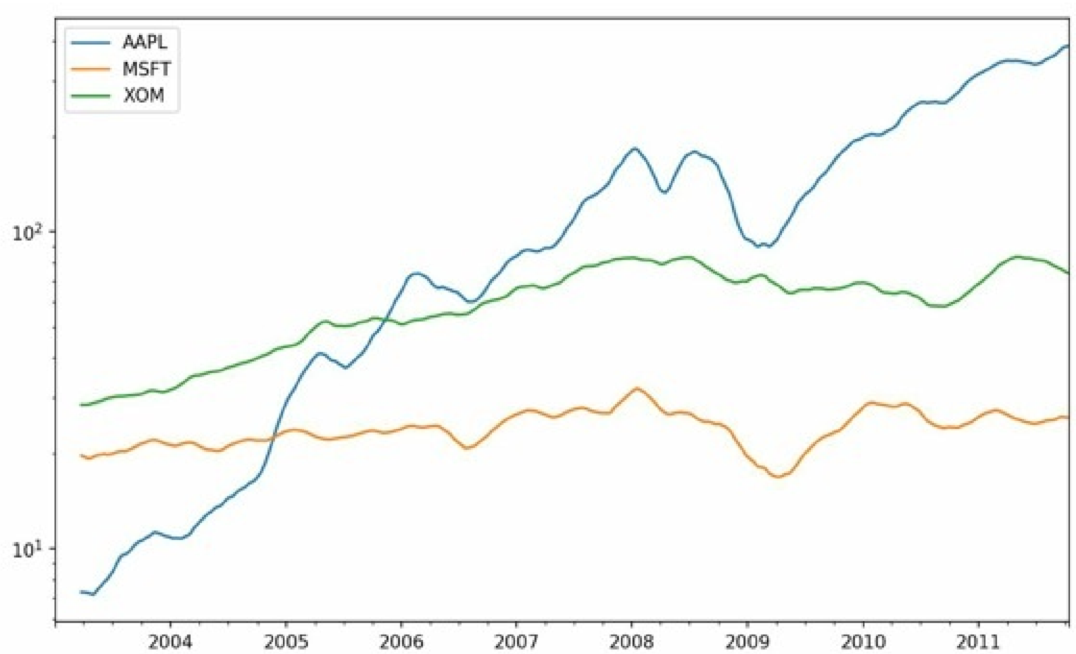


图11-6 各股价60日均线（对数Y轴）

rolling函数也可以接受一个指定固定大小时间补偿字符串，而不是一组时期。这样可以方便处理不规律的时间序列。这些字符串也可以传递给resample。例如，我们可以计算20天的滚动均值，如下所示：

```
In [247]: close_px.rolling('20D').mean()
Out[247]:
                  AAPL       MSFT        XOM
2003-01-02    7.400000  21.110000  29.220000
2003-01-03    7.425000  21.125000  29.230000
2003-01-06    7.433333  21.256667  29.473333
2003-01-07    7.432500  21.425000  29.342500
2003-01-08    7.402000  21.402000  29.240000
2003-01-09    7.391667  21.490000  29.273333
2003-01-10    7.387143  21.558571  29.238571
2003-01-13    7.378750  21.633750  29.197500
2003-01-14    7.370000  21.717778  29.194444
2003-01-15    7.355000  21.757000  29.152000
...                ...        ...        ...
2011-10-03  398.002143  25.890714  72.413571
```

```
2011-10-04  396.802143  25.807857  72.427143
2011-10-05  395.751429  25.729286  72.422857
2011-10-06  394.099286  25.673571  72.375714
2011-10-07  392.479333  25.712000  72.454667
2011-10-10  389.351429  25.602143  72.527857
2011-10-11  388.505000  25.674286  72.835000
2011-10-12  388.531429  25.810000  73.400714
2011-10-13  388.826429  25.961429  73.905000
2011-10-14  391.038000  26.048667  74.185333
[2292 rows x 3 columns]
```

指数加权函数

另一种使用固定大小窗口及相等权数观测值的办法是，定义一个衰减因子（decay factor）常量，以便使近期的观测值拥有更大的权数。衰减因子的定义方式有很多，比较流行的是使用时间间隔（span），它可以使结果兼容于窗口大小等于时间间隔的简单移动窗口（simple moving window）函数。

由于指数加权统计会赋予近期的观测值更大的权数，因此相对于等权统计，它能“适应”更快的变化。

除了rolling和expanding，pandas还有ewm运算符。下面这个例子对比了苹果公司股价的60日移动平均和span=60的指数加权移动平均（如图11-7所示）：

```
In [249]: aapl_px = close_px.AAPL['2006':'2007']

In [250]: ma60 = aapl_px.rolling(30, min_periods=20).

In [251]: ewma60 = aapl_px.ewm(span=30).mean()

In [252]: ma60.plot(style='k--', label='Simple MA')
Out[252]: <matplotlib.axes._subplots.AxesSubplot at 0

In [253]: ewma60.plot(style='k-', label='EW MA')
Out[253]: <matplotlib.axes._subplots.AxesSubplot at 0
```

```
In [254]: plt.legend()
```

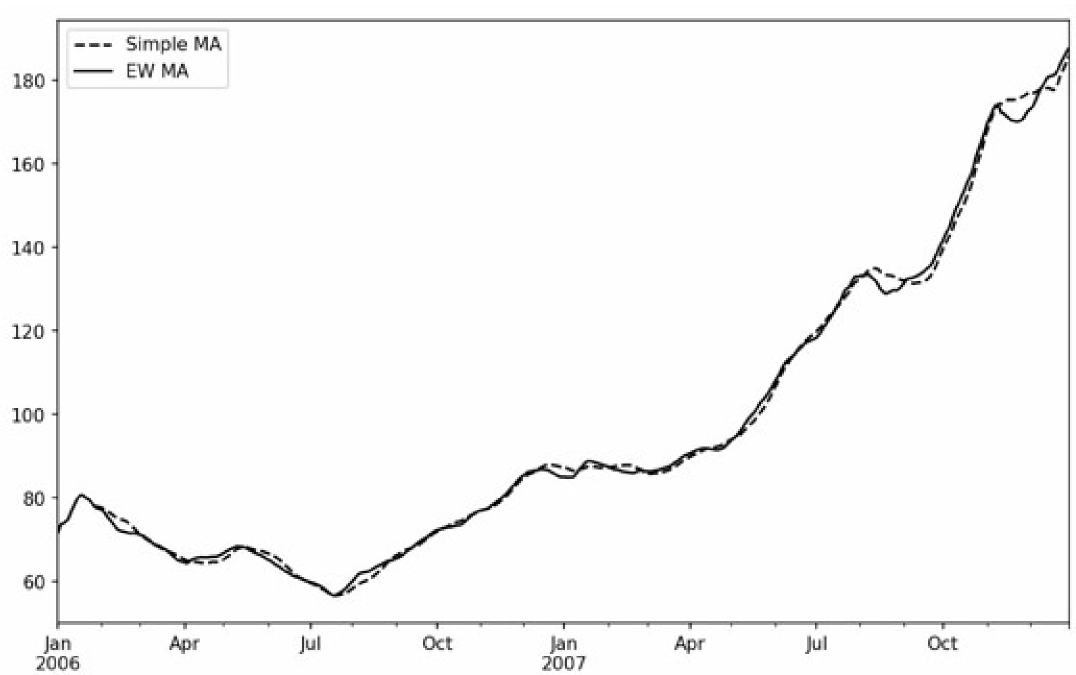


图11-7 简单移动平均与指数加权移动平均

二元移动窗口函数

有些统计运算（如相关系数和协方差）需要在两个时间序列上执行。例如，金融分析师常常对某只股票对某个参考指数（如标准普尔500指数）的相关系数感兴趣。要进行说明，我们先计算我们感兴趣的时间序列的百分数变化：

```
In [256]: spx_px = close_px_all['SPX']

In [257]: spx_rets = spx_px.pct_change()

In [258]: returns = close_px.pct_change()
```

调用rolling之后，corr聚合函数开始计算与spx_rets滚动相关系数（结果见图11-8）：

```
In [259]: corr = returns.AAPL.rolling(125, min_period
```

```
In [260]: corr.plot()
```

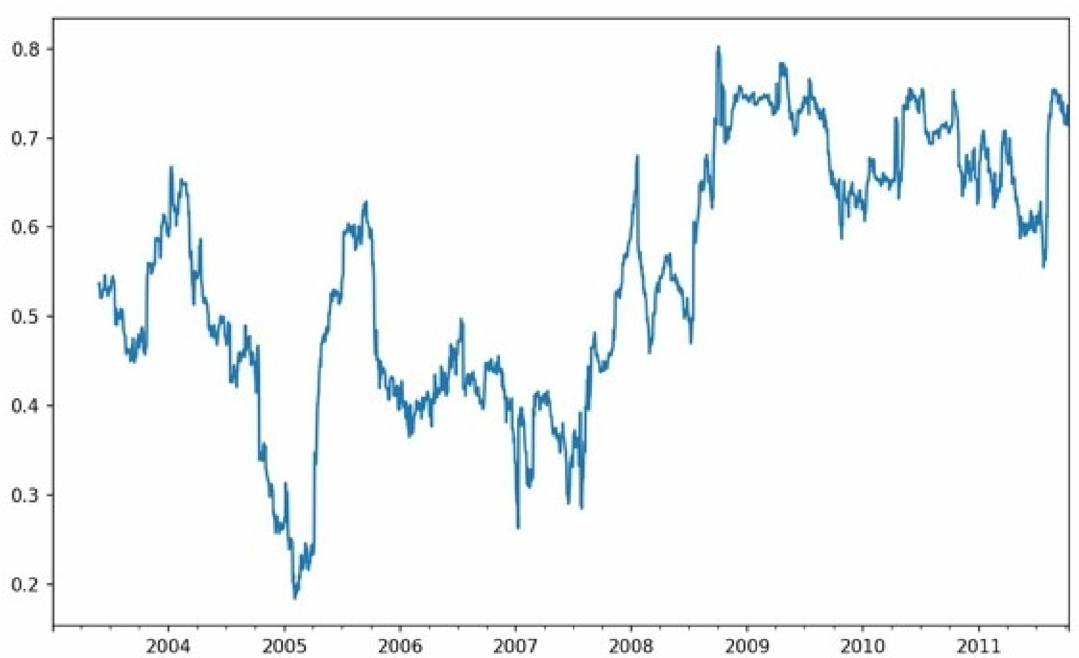


图11-8 AAPL 6个月的回报与标准普尔500指数的相关系数

假设你想要一次性计算多只股票与标准普尔500指数的相关系数。虽然编写一个循环并新建一个DataFrame不是什么难事，但比较啰嗦。其实，只需传入一个TimeSeries和一个DataFrame，rolling_corr就会自动计算TimeSeries（本例中就是spx_rets）与DataFrame各列的相关系数。结果如图11-9所示：

```
In [262]: corr = returns.rolling(125, min_periods=100

In [263]: corr.plot()
```



图11-9 3只股票6个月的回报与标准普尔500指数的相关系数

用户定义的移动窗口函数

rolling_apply函数使你能够在移动窗口上应用自己设计的数组函数。唯一要求的就是：该函数要能从数组的各个片段中产生单个值（即约简）。比如说，当我们用rolling(...).quantile(q)计算样本分位数时，可能对样本中特定值的百分等级感兴趣。scipy.stats.percentileofscore函数就能达到这个目的（结果见图11-10）：

```
In [265]: from scipy.stats import percentileofscore

In [266]: score_at_2percent = lambda x: percentileofs

In [267]: result = returns.AAPL.rolling(250).apply(sc

In [268]: result.plot()
```

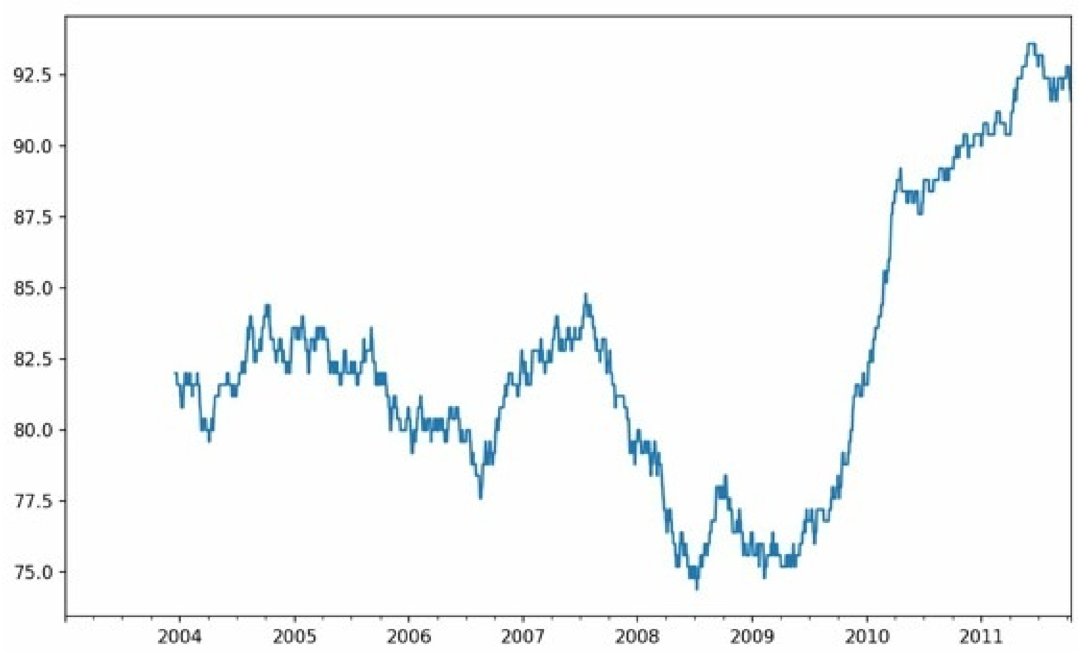


图11-10 AAPL 2%回报率的百分等级（一年窗口期）

如果你没安装SciPy，可以使用conda或pip安装。

11.8 总结

与前面章节接触的数据相比，时间序列数据要求不同类型的分析和数据转换工具。

在接下来的章节中，我们将学习一些高级的pandas方法和如何开始使用建模库statsmodels和scikit-learn。

第12章 pandas高级应用

前面的章节关注于不同类型的数据规整流程和NumPy、pandas与其它库的特点。随着时间的发展，pandas发展出了更多适合高级用户的功能。本章就要深入学习pandas的高级功能。

12.1 分类数据

这一节介绍的是pandas的分类类型。我会向你展示通过使用它，提高性能和内存的使用率。我还会介绍一些在统计和机器学习中使用分类数据的工具。

背景和目的

表中的一列通常会有重复的包含不同值的小集合的情况。我们已经学过了unique和value_counts，它们可以从数组提取出不同的值，并分别计算频率：

```
In [10]: import numpy as np; import pandas as pd

In [11]: values = pd.Series(['apple', 'orange', 'appl
   ....:                     'apple'] * 2)

In [12]: values
Out[12]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
dtype: object

In [13]: pd.unique(values)
Out[13]: array(['apple', 'orange'], dtype=object)
```

```
In [14]: pd.value_counts(values)
Out[14]:
apple     6
orange    2
dtype: int64
```

许多数据系统（数据仓库、统计计算或其它应用）都发展出了特定的表征重复值的方法，以进行高效的存储和计算。在数据仓库中，最好的方法是使用所谓的包含不同值得维表(Dimension Table)，将主要的参数存储为引用维表整数键：

```
In [15]: values = pd.Series([0, 1, 0, 0] * 2)

In [16]: dim = pd.Series(['apple', 'orange'])

In [17]: values
Out[17]:
0    0
1    1
2    0
3    0
4    0
5    1
6    0
7    0
dtype: int64

In [18]: dim
Out[18]:
0     apple
1    orange
dtype: object
```

可以使用take方法存储原始的字符串Series：

```
In [19]: dim.take(values)
Out[19]:
0     apple
1    orange
0     apple
0     apple
0     apple
1    orange
0     apple
0     apple
dtype: object
```

这种用整数表示的方法称为分类或字典编码表示法。不同值得数组称为分类、字典或数据级。本书中，我们使用分类的说法。表示分类的整数值称为分类编码或简单地称为编码。

分类表示可以在进行分析时大大的提高性能。你也可以在保持编码不变的情况下，对分类进行转换。一些相对简单的转变例子包括：

- 重命名分类。
- 加入一个新的分类，不改变已经存在的分类的顺序或位置。

pandas的分类类型

pandas有一个特殊的分类类型，用于保存使用整数分类表示法的数据。看一个之前的Series例子：

```
In [20]: fruits = ['apple', 'orange', 'apple', 'apple

In [21]: N = len(fruits)

In [22]: df = pd.DataFrame({'fruit': fruits,
   ....:                    'basket_id': np.arange(N)
   ....:                    'count': np.random.randin
   ....:                    'weight': np.random.unifo
   ....:                   columns=['basket_id', 'fru
```

```
In [23]: df
Out[23]:
   basket_id   fruit  count    weight
0          0   apple      5  3.858058
1          1  orange      8  2.612708
2          2   apple      4  2.995627
3          3   apple      7  2.614279
4          4   apple     12  2.990859
5          5  orange      8  3.845227
6          6   apple      5  0.033553
7          7   apple      4  0.425778
```

这里，df['fruit']是一个Python字符串对象的数组。我们可以通过调用它，将它转变为分类：

```
In [24]: fruit_cat = df['fruit'].astype('category')

In [25]: fruit_cat
Out[25]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

fruit_cat的值不是NumPy数组，而是一个pandas.Categorical实例：

```
In [26]: c = fruit_cat.values
```

```
In [27]: type(c)
Out[27]: pandas.core.categorical.Categorical
```

分类对象有categories和codes属性：

```
In [28]: c.categories
Out[28]: Index(['apple', 'orange'], dtype='object')

In [29]: c.codes
Out[29]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

你可将DataFrame的列通过分配转换结果，转换为分类：

```
In [30]: df['fruit'] = df['fruit'].astype('category')

In [31]: df.fruit
Out[31]:
0     apple
1    orange
2     apple
3     apple
4     apple
5    orange
6     apple
7     apple
Name: fruit, dtype: category
Categories (2, object): [apple, orange]
```

你还可以从其它Python序列直接创建pandas.Categorical：

```
In [32]: my_categories = pd.Categorical(['foo', 'bar'

In [33]: my_categories
Out[33]:
[foo, bar, baz, foo, bar]
Categories (3, object): [bar, baz, foo]
```

如果你已经从其它源获得了分类编码，你还可以使用from_codes构造器：

```
In [34]: categories = ['foo', 'bar', 'baz']

In [35]: codes = [0, 1, 2, 0, 0, 1]

In [36]: my_cats_2 = pd.Categorical.from_codes(codes,

In [37]: my_cats_2
Out[37]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo, bar, baz]
```

与显示指定不同，分类变换不认定指定的分类顺序。因此取决于输入数据的顺序，categories数组的顺序会不同。当使用from_codes或其它的构造器时，你可以指定分类一个有意义的顺序：

```
In [38]: ordered_cat = pd.Categorical.from_codes(code
   ....:                                         orde

In [39]: ordered_cat
Out[39]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

输出[foo < bar < baz]指明'foo'位于'bar'的前面，以此类推。无序的分类实例可以通过as_ordered排序：

```
In [40]: my_cats_2.as_ordered()
Out[40]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

最后要注意，分类数据不需要字符串，尽管我仅仅展示了字符串的例子。分类数组可以包括任意不可变类型。

用分类进行计算

与非编码版本（比如字符串数组）相比，使用pandas的Categorical有些类似。某些pandas组件，比如groupby函数，更适合进行分类。还有一些函数可以使用有序标志位。

来看一些随机的数值数据，使用pandas.qcut面元函数。它会返回pandas.Categorical，我们之前使用过pandas.cut，但没解释分类是如何工作的：

```
In [41]: np.random.seed(12345)

In [42]: draws = np.random.randn(1000)

In [43]: draws[:5]
Out[43]: array([-0.2047,  0.4789, -0.5194, -0.5557,  
```

计算这个数据的分位面元，提取一些统计信息：

```
In [44]: bins = pd.qcut(draws, 4)

In [45]: bins
Out[45]:
[(-0.684, -0.0101], (-0.0101, 0.63], (-0.684, -0.0101
 3.928], ..., (-0.0101, 0.63], (-0.684, -0.0101], (-2
], (0.63, 3.928]]
Length: 1000
Categories (4, interval[float64]): [(-2.95, -0.684] <
1, 0.63] <
                                    (0.63, 3.928]]
```

虽然有用，确切的样本分位数与分位的名称相比，不利于生成汇总。我们可以使用labels参数qcut，实现目的：

```
In [46]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2',

In [47]: bins
Out[47]:
[Q2, Q3, Q2, Q2, Q4, ..., Q3, Q2, Q1, Q3, Q4]
Length: 1000
Categories (4, object): [Q1 < Q2 < Q3 < Q4]

In [48]: bins.codes[:10]
Out[48]: array([1, 2, 1, 1, 3, 3, 2, 2, 3, 3], dtype=
```

加上标签的面元分类不包含数据面元边界的信息，因此可以使用groupby提取一些汇总信息：

```
In [49]: bins = pd.Series(bins, name='quartile')

In [50]: results = (pd.Series(draws)
   ....:             .groupby(bins)
   ....:             .agg(['count', 'min', 'max'])
   ....:             .reset_index())

In [51]: results
Out[51]:
  quartile  count       min       max
0       Q1    250 -2.949343 -0.685484
1       Q2    250 -0.683066 -0.010115
2       Q3    250 -0.010032  0.628894
3       Q4    250  0.634238  3.927528
```

分位数列保存了原始的面元分类信息，包括排序：

```
In [52]: results['quartile']
Out[52]:
0    Q1
1    Q2
```

```
2    Q3
3    Q4
Name: quartile, dtype: category
Categories (4, object): [Q1 < Q2 < Q3 < Q4]
```

用分类提高性能

如果你是在一个特定数据集上做大量分析，将其转换为分类可以极大地提高效率。DataFrame列的分类使用的内存通常少的多。来看一些包含一千万元素的Series，和一些不同的分类：

```
In [53]: N = 10000000

In [54]: draws = pd.Series(np.random.randn(N))

In [55]: labels = pd.Series(['foo', 'bar', 'baz', 'qu
```

现在，将标签转换为分类：

```
In [56]: categories = labels.astype('category')
```

这时，可以看到标签使用的内存远比分类多：

```
In [57]: labels.memory_usage()
Out[57]: 80000080

In [58]: categories.memory_usage()
Out[58]: 10000272
```

转换为分类不是没有代价的，但这是一次性的代价：

```
In [59]: %time _ = labels.astype('category')
CPU times: user 490 ms, sys: 240 ms, total: 730 ms
Wall time: 726 ms
```

GroupBy操作明显比分类快，是因为底层的算法使用整数编码数

组，而不是字符串数组。

分类方法

包含分类数据的Series有一些特殊的方法，类似于Series.str字符串方法。它还提供了方便的分类和编码的使用方法。看下面的Series：

```
In [60]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)

In [61]: cat_s = s.astype('category')

In [62]: cat_s
Out[62]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (4, object): [a, b, c, d]
```

特别的cat属性提供了分类方法的入口：

```
In [63]: cat_s.cat.codes
Out[63]:
0    0
1    1
2    2
3    3
4    0
5    1
6    2
7    3
```

```
dtype: int8

In [64]: cat_s.cat.categories
Out[64]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

假设我们知道这个数据的实际分类集，超出了数据中的四个值。我们可以使用set_categories方法改变它们：

```
In [65]: actual_categories = ['a', 'b', 'c', 'd', 'e'

In [66]: cat_s2 = cat_s.cat.set_categories(actual_cat

In [67]: cat_s2
Out[67]:
0    a
1    b
2    c
3    d
4    a
5    b
6    c
7    d
dtype: category
Categories (5, object): [a, b, c, d, e]
```

虽然数据看起来没变，新的分类将反映在它们的操作中。例如，如果有的话，value_counts表示分类：

```
In [68]: cat_s.value_counts()
Out[68]:
d    2
c    2
b    2
a    2
dtype: int64
```

```
In [69]: cat_s2.value_counts()
Out[69]:
d    2
c    2
b    2
a    2
e    0
dtype: int64
```

在打数据集中，分类经常作为节省内存和高性能的便捷工具。过滤完大DataFrame或Series之后，许多分类可能不会出现在数据中。我们可以使用remove_unused_categories方法删除没看到的分类：

```
In [70]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]

In [71]: cat_s3
Out[71]:
0    a
1    b
4    a
5    b
dtype: category
Categories (4, object): [a, b, c, d]

In [72]: cat_s3.cat.remove_unused_categories()
Out[72]:
0    a
1    b
4    a
5    b
dtype: category
Categories (2, object): [a, b]
```

表12-1列出了可用的分类方法。

方法	说明
add_categories	在已存在的分类后面添加新的（未使用的）分类
as_ordered	使分类有序
as_unordered	使分类无序
remove_categories	移除分类，设置任何被移除的值为 null
remove_unused_categories	移除任意不出现在数据中的分类值
rename_categories	用指定的新分类的名字替换分类；不能改变分类的数目
reorder_categories	与 rename_categories 很像，但是可以改变结果，使分类有序
set_categories	用指定的新分类的名字替换分类；可以添加或删除分类

表12-1 pandas的Series的分类方法

为建模创建虚拟变量

当你使用统计或机器学习工具时，通常会将分类数据转换为虚拟变量，也称为one-hot编码。这包括创建一个不同类别的列的DataFrame；这些列包含给定分类的1s，其它为0。

看前面的例子：

```
In [73]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2,
```

前面的第7章提到过，pandas.get_dummies函数可以转换这个以为分类数据为包含虚拟变量的DataFrame：

```
In [74]: pd.get_dummies(cat_s)
Out[74]:
   a  b  c  d
0  1  0  0  0
1  0  1  0  0
2  0  0  1  0
3  0  0  0  1
4  1  0  0  0
5  0  1  0  0
6  0  0  1  0
```

```
7  0  0  0  1
```

12.2 GroupBy高级应用

尽管我们在第10章已经深度学习了Series和DataFrame的Groupby方法，还有一些方法也是很有用的。

分组转换和“解封”GroupBy

在第10章，我们在分组操作中学习了apply方法，进行转换。还有另一个transform方法，它与apply很像，但是对使用的函数有一定限制：

- 它可以产生向分组形状广播标量值
- 它可以产生一个和输入组形状相同的对象
- 它不能修改输入

来看一个简单的例子：

```
In [75]: df = pd.DataFrame({'key': ['a', 'b', 'c'] * 
   ....:                            'value': np.arange(12.)})

In [76]: df
Out[76]: 
   key  value
0    a    0.0
1    b    1.0
2    c    2.0
3    a    3.0
4    b    4.0
5    c    5.0
6    a    6.0
7    b    7.0
8    c    8.0
9    a    9.0
10   b   10.0
11   c   11.0
```

按键进行分组：

```
In [77]: g = df.groupby('key').value

In [78]: g.mean()
Out[78]:
key
a    4.5
b    5.5
c    6.5
Name: value, dtype: float64
```

假设我们想产生一个和df['value']形状相同的Series，但值替换为按键分组的平均值。我们可以传递函数lambda x: x.mean()进行转换：

```
In [79]: g.transform(lambda x: x.mean())
Out[79]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

对于内置的聚合函数，我们可以传递一个字符串假名作为GroupBy的agg方法：

```
In [80]: g.transform('mean')
```

```
Out[80]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
Name: value, dtype: float64
```

与apply类似，transform的函数会返回Series，但是结果必须与输入大小相同。举个例子，我们可以用lambda函数将每个分组乘以2：

```
In [81]: g.transform(lambda x: x * 2)
Out[81]:
0      0.0
1      2.0
2      4.0
3      6.0
4      8.0
5     10.0
6     12.0
7     14.0
8     16.0
9     18.0
10    20.0
11    22.0
Name: value, dtype: float64
```

再举一个复杂的例子，我们可以计算每个分组的降序排名：

```
In [82]: g.transform(lambda x: x.rank(ascending=False
Out[82]:
0     4.0
1     4.0
2     4.0
3     3.0
4     3.0
5     3.0
6     2.0
7     2.0
8     2.0
9     1.0
10    1.0
11    1.0
Name: value, dtype: float64
```

看一个由简单聚合构造的的分组转换函数：

```
def normalize(x):
    return (x - x.mean()) / x.std()
```

我们用transform或apply可以获得等价的结果：

```
In [84]: g.transform(normalize)
Out[84]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
```

```
11    1.161895
Name: value, dtype: float64

In [85]: g.apply(normalize)
Out[85]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

内置的聚合函数，比如mean或sum，通常比apply函数快，也比transform快。这允许我们进行一个所谓的解封（unwrapped）分组操作：

```
In [86]: g.transform('mean')
Out[86]:
0     4.5
1     5.5
2     6.5
3     4.5
4     5.5
5     6.5
6     4.5
7     5.5
8     6.5
9     4.5
10    5.5
11    6.5
```

```
Name: value, dtype: float64

In [87]: normalized = (df['value'] - g.transform('mea

In [88]: normalized
Out[88]:
0    -1.161895
1    -1.161895
2    -1.161895
3    -0.387298
4    -0.387298
5    -0.387298
6     0.387298
7     0.387298
8     0.387298
9     1.161895
10    1.161895
11    1.161895
Name: value, dtype: float64
```

解封分组操作可能包括多个分组聚合，但是矢量化操作还是会带来收益。

分组的时间重采样

对于时间序列数据，resample方法从语义上是一个基于内在时间的分组操作。下面是一个示例表：

```
In [89]: N = 15

In [90]: times = pd.date_range('2017-05-20 00:00', fr

In [91]: df = pd.DataFrame({'time': times,
   ....:                    'value': np.arange(N)})

In [92]: df
Out[92]:
```

```
                    time  value
0  2017-05-20 00:00:00      0
1  2017-05-20 00:01:00      1
2  2017-05-20 00:02:00      2
3  2017-05-20 00:03:00      3
4  2017-05-20 00:04:00      4
5  2017-05-20 00:05:00      5
6  2017-05-20 00:06:00      6
7  2017-05-20 00:07:00      7
8  2017-05-20 00:08:00      8
9  2017-05-20 00:09:00      9
10 2017-05-20 00:10:00     10
11 2017-05-20 00:11:00     11
12 2017-05-20 00:12:00     12
13 2017-05-20 00:13:00     13
14 2017-05-20 00:14:00     14
```

这里，我们可以用time作为索引，然后重采样：

```
In [93]: df.set_index('time').resample('5min').count(
Out[93]:
                     value
time
2017-05-20 00:00:00      5
2017-05-20 00:05:00      5
2017-05-20 00:10:00      5
```

假设DataFrame包含多个时间序列，用一个额外的分组键的列进行标记：

```
In [94]: df2 = pd.DataFrame({'time': times.repeat(3),
   ....:                     'key': np.tile(['a', 'b'
   ....:                     'value': np.arange(N * 3

In [95]: df2[:7]
Out[95]:
```

```
  key                 time  value
0   a 2017-05-20 00:00:00    0.0
1   b 2017-05-20 00:00:00    1.0
2   c 2017-05-20 00:00:00    2.0
3   a 2017-05-20 00:01:00    3.0
4   b 2017-05-20 00:01:00    4.0
5   c 2017-05-20 00:01:00    5.0
6   a 2017-05-20 00:02:00    6.0
```

要对每个key值进行相同的重采样，我们引入pandas.TimeGrouper对象：

```
In [96]: time_key = pd.TimeGrouper('5min')
```

我们然后设定时间索引，用key和time_key分组，然后聚合：

```
In [97]: resampled = (df2.set_index('time')
   ....:              .groupby(['key', time_key])
   ....:              .sum())

In [98]: resampled
Out[98]:
                         value
key time
a   2017-05-20 00:00:00   30.0
    2017-05-20 00:05:00  105.0
    2017-05-20 00:10:00  180.0
b   2017-05-20 00:00:00   35.0
    2017-05-20 00:05:00  110.0
    2017-05-20 00:10:00  185.0
c   2017-05-20 00:00:00   40.0
    2017-05-20 00:05:00  115.0
    2017-05-20 00:10:00  190.0

In [99]: resampled.reset_index()
Out[99]:
```

```
key                 time  value
0   a 2017-05-20 00:00:00   30.0
1   a 2017-05-20 00:05:00  105.0
2   a 2017-05-20 00:10:00  180.0
3   b 2017-05-20 00:00:00   35.0
4   b 2017-05-20 00:05:00  110.0
5   b 2017-05-20 00:10:00  185.0
6   c 2017-05-20 00:00:00   40.0
7   c 2017-05-20 00:05:00  115.0
8   c 2017-05-20 00:10:00  190.0
```

使用TimeGrouper的限制是时间必须是Series或DataFrame的索引。

12.3 链式编程技术

当对数据集进行一系列变换时，你可能发现创建的多个临时变量其实并没有在分析中用到。看下面的例子：

```
df = load_data()
df2 = df[df['col2'] < 0]
df2['col1_demeaned'] = df2['col1'] - df2['col1'].mean
result = df2.groupby('key').col1_demeaned.std()
```

虽然这里没有使用真实的数据，这个例子却指出了一些新方法。首先，DataFrame.assign方法是一个df[k] = v形式的函数式的列分配方法。它不是就地修改对象，而是返回新的修改过的DataFrame。因此，下面的语句是等价的：

```
# Usual non-functional way
df2 = df.copy()
df2['k'] = v

# Functional assign way
df2 = df.assign(k=v)
```

就地分配可能会比assign快，但是assign可以方便地进行链式编程：

```
result = (df2.assign(col1_demeaned=df2.col1 - df2.col2
          .groupby('key')
          .col1_demeaned.std())
```

我使用外括号，这样便于添加换行符。

使用链式编程时要注意，你可能会需要涉及临时对象。在前面的例子中，我们不能使用load_data的结果，直到它被赋值给临时变量df。为了这么做，assign和许多其它pandas函数可以接收类似函数的参数，即可调用对象（callable）。为了展示可调用对象，看一个前面例子的片段：

```
df = load_data()
df2 = df[df['col2'] < 0]
```

它可以重写为：

```
df = (load_data()
      [lambda x: x['col2'] < 0])
```

这里，load_data的结果没有赋值给某个变量，因此传递到[ ]的函数在这一步被绑定到了对象。

我们可以把整个过程写为一个单链表达式：

```
result = (load_data()
          [lambda x: x.col2 < 0]
          .assign(col1_demeaned=lambda x: x.col1 - x.c
          .groupby('key')
          .col1_demeaned.std())
```

是否将代码写成这种形式只是习惯而已，将它分开成若干步可以

提高可读性。

管道方法

你可以用Python内置的pandas函数和方法，用带有可调用对象的链式编程做许多工作。但是，有时你需要使用自己的函数，或是第三方库的函数。这时就要用到管道方法。

看下面的函数调用：

```
a = f(df, arg1=v1)
b = g(a, v2, arg3=v3)
c = h(b, arg4=v4)
```

当使用接收、返回Series或DataFrame对象的函数式，你可以调用pipe将其重写：

```
result = (df.pipe(f, arg1=v1)
          .pipe(g, v2, arg3=v3)
          .pipe(h, arg4=v4))
```

f(df)和df.pipe(f)是等价的，但是pipe使得链式声明更容易。

pipe的另一个有用的地方是提炼操作为可复用的函数。看一个从列减去分组方法的例子：

```
g = df.groupby(['key1', 'key2'])
df['col1'] = df['col1'] - g.transform('mean')
```

假设你想转换多列，并修改分组的键。另外，你想用链式编程做这个转换。下面就是一个方法：

```
def group_demean(df, by, cols):
    result = df.copy()
    g = df.groupby(by)
    for c in cols:
        result[c] = df[c] - g[c].transform('mean')
```

```
    return result
```

然后可以写为：

```
result = (df[df.col1 < 0]
          .pipe(group_demean, ['key1', 'key2'], ['col1
```

12.4 总结

和其它许多开源项目一样，pandas仍然在不断的变化和进步中。和本书中其它地方一样，这里的重点是放在接下来几年不会发生什么改变且稳定的功能。

为了深入学习pandas的知识，我建议你学习官方文档，并阅读开发团队发布的更新文档。我们还邀请你加入pandas的开发工作：修改bug、创建新功能、完善文档。

第13章 Python建模库介绍

本书中，我已经介绍了Python数据分析的编程基础。因为数据分析师和科学家总是在数据规整和准备上花费大量时间，这本书的重点在于掌握这些功能。

开发模型选用什么库取决于应用本身。许多统计问题可以用简单方法解决，比如普通的最小二乘回归，其它问题可能需要复杂的机器学习方法。幸运的是，Python已经成为了运用这些分析方法的语言之一，因此读完此书，你可以探索许多工具。

本章中，我会回顾一些pandas的特点，在你胶着于pandas数据规整和模型拟合和评分时，它们可能派上用场。然后我会简短介绍两个流行的建模工具，statsmodels和scikit-learn。这二者每个都值得再写一本书，我就不做全面的介绍，而是建议你学习两个项目的线上文档和其它基于Python的数据科学、统计和机器学习的书籍。

13.1 pandas与模型代码的接口

模型开发的通常工作流是使用pandas进行数据加载和清洗，然后切换到建模库进行建模。开发模型的重要一环是机器学习中的“特征工程”。它可以描述从原始数据集中提取信息的任何数据转换或分析，这些数据集可能在建模中有用。本书中学习的数据聚合和GroupBy工具常用于特征工程中。

优秀的特征工程超出了本书的范围，我会尽量直白地介绍一些用于数据操作和建模切换的方法。

pandas与其它分析库通常是靠NumPy的数组联系起来的。将DataFrame转换为NumPy数组，可以使用.values属性：

```
In [10]: import pandas as pd

In [11]: import numpy as np
```

```
In [12]: data = pd.DataFrame({
   ....:     'x0': [1, 2, 3, 4, 5],
   ....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [13]: data
Out[13]:
   x0    x1    y
0   1  0.01 -1.5
1   2 -0.01  0.0
2   3  0.25  3.6
3   4 -4.10  1.3
4   5  0.00 -2.0

In [14]: data.columns
Out[14]: Index(['x0', 'x1', 'y'], dtype='object')

In [15]: data.values
Out[15]:
array([[ 1.  ,  0.01, -1.5 ],
       [ 2.  , -0.01,  0.  ],
       [ 3.  ,  0.25,  3.6 ],
       [ 4.  , -4.1 ,  1.3 ],
       [ 5.  ,  0.  , -2.  ]])
```

要转换回DataFrame，可以传递一个二维ndarray，可带有列名：

```
In [16]: df2 = pd.DataFrame(data.values, columns=['on

In [17]: df2
Out[17]:
   one   two  three
0  1.0  0.01   -1.5
1  2.0 -0.01    0.0
2  3.0  0.25    3.6
3  4.0 -4.10    1.3
```

```
4  5.0  0.00   -2.0
```

> 笔记：最好当数据是均匀的时候使用.values属性。例如，全是数值类型。如果数据是不均匀的，结果会是Python对象的ndarray：
>
> ```
> In [18]: df3 = data.copy()
>
> In [19]: df3['strings'] = ['a', 'b', 'c', 'd', 'e']
>
> In [20]: df3
> Out[20]:
>   x0    x1    y strings
> 0   1  0.01 -1.5       a
> 1   2 -0.01  0.0       b
> 2   3  0.25  3.6       c
> 3   4 -4.10  1.3       d
> 4   5  0.00 -2.0       e
>
> In [21]: df3.values
> Out[21]:
> array([[1, 0.01, -1.5, 'a'],
>        [2, -0.01, 0.0, 'b'],
>        [3, 0.25, 3.6, 'c'],
>        [4, -4.1, 1.3, 'd'],
>        [5, 0.0, -2.0, 'e']], dtype=object)
> ```

对于一些模型，你可能只想使用列的子集。我建议你使用loc，用values作索引：

```
In [22]: model_cols = ['x0', 'x1']

In [23]: data.loc[:, model_cols].values
Out[23]:
array([[ 1.  ,  0.01],
```

```
       [ 2.  , -0.01],
       [ 3.  ,  0.25],
       [ 4.  , -4.1 ],
       [ 5.  ,  0.  ]])
```

一些库原生支持pandas，会自动完成工作：从DataFrame转换到NumPy，将模型的参数名添加到输出表的列或Series。其它情况，你可以手工进行“元数据管理”。

在第12章，我们学习了pandas的Categorical类型和pandas.get_dummies函数。假设数据集中有一个非数值列：

```
In [24]: data['category'] = pd.Categorical(['a', 'b',
   ....:                                        categories

In [25]: data
Out[25]:
   x0    x1    y category
0   1  0.01 -1.5        a
1   2 -0.01  0.0        b
2   3  0.25  3.6        a
3   4 -4.10  1.3        a
4   5  0.00 -2.0        b
```

如果我们想替换category列为虚变量，我们可以创建虚变量，删除category列，然后添加到结果：

```
In [26]: dummies = pd.get_dummies(data.category, pref

In [27]: data_with_dummies = data.drop('category', ax

In [28]: data_with_dummies
Out[28]:
   x0    x1    y  category_a  category_b
0   1  0.01 -1.5           1           0
1   2 -0.01  0.0           0           1
```

```
2   3  0.25  3.6            1            0
3   4 -4.10  1.3            1            0
4   5  0.00 -2.0            0            1
```

用虚变量拟合某些统计模型会有一些细微差别。当你不只有数字列时，使用Patsy（下一节的主题）可能更简单，更不容易出错。

13.2 用Patsy创建模型描述

Patsy是Python的一个库，使用简短的字符串“公式语法”描述统计模型（尤其是线性模型），可能是受到了R和S统计编程语言的公式语法的启发。

Patsy适合描述statsmodels的线性模型，因此我会关注于它的主要特点，让你尽快掌握。Patsy的公式是一个特殊的字符串语法，如下所示：

```
y ~ x0 + x1
```

a+b不是将a与b相加的意思，而是为模型创建的设计矩阵。patsy.dmatrices函数接收一个公式字符串和一个数据集（可以是DataFrame或数组的字典），为线性模型创建设计矩阵：

```
In [29]: data = pd.DataFrame({
   ....:     'x0': [1, 2, 3, 4, 5],
   ....:     'x1': [0.01, -0.01, 0.25, -4.1, 0.],
   ....:     'y': [-1.5, 0., 3.6, 1.3, -2.]})

In [30]: data
Out[30]:
   x0    x1    y
0   1  0.01 -1.5
1   2 -0.01  0.0
2   3  0.25  3.6
3   4 -4.10  1.3
```

```
4   5  0.00 -2.0

In [31]: import patsy

In [32]: y, X = patsy.dmatrices('y ~ x0 + x1', data)
```

现在有：

```
In [33]: y
Out[33]:
DesignMatrix with shape (5, 1)
     y
  -1.5
   0.0
   3.6
   1.3
  -2.0
  Terms:
    'y' (column 0)

In [34]: X
Out[34]:
DesignMatrix with shape (5, 3)
  Intercept  x0     x1
          1   1   0.01
          1   2  -0.01
          1   3   0.25
          1   4  -4.10
          1   5   0.00
  Terms:
    'Intercept' (column 0)
    'x0' (column 1)
    'x1' (column 2)
```

这些Patsy的DesignMatrix实例是NumPy的ndarray，带有附加元数据：

```
In [35]: np.asarray(y)
Out[35]:
array([[-1.5],
       [ 0. ],
       [ 3.6],
       [ 1.3],
       [-2. ]])

In [36]: np.asarray(X)
Out[36]:
array([[ 1.  ,  1.  ,  0.01],
       [ 1.  ,  2.  , -0.01],
       [ 1.  ,  3.  ,  0.25],
       [ 1.  ,  4.  , -4.1 ],
       [ 1.  ,  5.  ,  0.  ]])
```

你可能想Intercept是哪里来的。这是线性模型（比如普通最小二乘胡桂）的惯例用法。添加 +0 到模型可以不显示intercept：

```
In [37]: patsy.dmatrices('y ~ x0 + x1 + 0', data)[1]
Out[37]:
DesignMatrix with shape (5, 2)
  x0     x1
   1   0.01
   2  -0.01
   3   0.25
   4  -4.10
   5   0.00
  Terms:
    'x0' (column 0)
    'x1' (column 1)
```

Patsy对象可以直接传递到算法（比如numpy.linalg.lstsq）中，它执行普通最小二乘回归：

```
In [38]: coef, resid, _, _ = np.linalg.lstsq(X, y)
```

模型的元数据保留在design_info属性中，因此你可以重新附加列名到拟合系数，以获得一个Series，例如：

```
In [39]: coef
Out[39]:
array([[ 0.3129],
       [-0.0791],
       [-0.2655]])

In [40]: coef = pd.Series(coef.squeeze(), index=X.des

In [41]: coef
Out[41]:
Intercept    0.312910
x0          -0.079106
x1          -0.265464
dtype: float64
```

用Patsy公式进行数据转换

你可以将Python代码与patsy公式结合。在评估公式时，库将尝试查找在封闭作用域内使用的函数：

```
In [42]: y, X = patsy.dmatrices('y ~ x0 + np.log(np.a

In [43]: X
Out[43]:
DesignMatrix with shape (5, 3)
  Intercept  x0  np.log(np.abs(x1) + 1)
          1   1                 0.00995
          1   2                 0.00995
          1   3                 0.22314
          1   4                 1.62924
          1   5                 0.00000
  Terms:
    'Intercept' (column 0)
```

```
    'x0' (column 1)
    'np.log(np.abs(x1) + 1)' (column 2)
```

常见的变量转换包括标准化（平均值为0，方差为1）和中心化（减去平均值）。Patsy有内置的函数进行这样的工作：

```
In [44]: y, X = patsy.dmatrices('y ~ standardize(x0) 

In [45]: X
Out[45]:
DesignMatrix with shape (5, 3)
  Intercept  standardize(x0)  center(x1)
          1         -1.41421        0.78
          1         -0.70711        0.76
          1          0.00000        1.02
          1          0.70711       -3.33
          1          1.41421        0.77
  Terms:
    'Intercept' (column 0)
    'standardize(x0)' (column 1)
    'center(x1)' (column 2)
```

作为建模的一步，你可能拟合模型到一个数据集，然后用另一个数据集评估模型。另一个数据集可能是剩余的部分或是新数据。当执行中心化和标准化转变，用新数据进行预测要格外小心。因为你必须使用平均值或标准差转换新数据集，这也称作状态转换。

patsy.build_design_matrices函数可以应用于转换新数据，使用原始样本数据集的保存信息：

```
In [46]: new_data = pd.DataFrame({
   ....:     'x0': [6, 7, 8, 9],
   ....:     'x1': [3.1, -0.5, 0, 2.3],
   ....:     'y': [1, 2, 3, 4]})
```

```
In [47]: new_X = patsy.build_design_matrices([X.desig

In [48]: new_X
Out[48]:
[DesignMatrix with shape (4, 3)
   Intercept  standardize(x0)  center(x1)
           1          2.12132        3.87
           1          2.82843        0.27
           1          3.53553        0.77
           1          4.24264        3.07
   Terms:
     'Intercept' (column 0)
     'standardize(x0)' (column 1)
     'center(x1)' (column 2)]
```

因为Patsy中的加号不是加法的意义，当你按照名称将数据集的列相加时，你必须用特殊I函数将它们封装起来：

```
In [49]: y, X = patsy.dmatrices('y ~ I(x0 + x1)', data

In [50]: X
Out[50]:
DesignMatrix with shape (5, 2)
  Intercept  I(x0 + x1)
          1        1.01
          1        1.99
          1        3.25
          1       -0.10
          1        5.00
  Terms:
    'Intercept' (column 0)
    'I(x0 + x1)' (column 1)
```

Patsy的patsy.builtins模块还有一些其它的内置转换。请查看线

上文档。

分类数据有一个特殊的转换类，下面进行讲解。

分类数据和Patsy

非数值数据可以用多种方式转换为模型设计矩阵。完整的讲解超出了本书范围，最好和统计课一起学习。

当你在Patsy公式中使用非数值数据，它们会默认转换为虚变量。如果有截距，会去掉一个，避免共线性：

```
In [51]: data = pd.DataFrame({
   ....:     'key1': ['a', 'a', 'b', 'b', 'a', 'b', '
   ....:     'key2': [0, 1, 0, 1, 0, 1, 0, 0],
   ....:     'v1': [1, 2, 3, 4, 5, 6, 7, 8],
   ....:     'v2': [-1, 0, 2.5, -0.5, 4.0, -1.2, 0.2,
   ....: })

In [52]: y, X = patsy.dmatrices('v2 ~ key1', data)

In [53]: X
Out[53]:
DesignMatrix with shape (8, 2)
  Intercept  key1[T.b]
          1          0
          1          0
          1          1
          1          1
          1          0
          1          1
          1          0
          1          1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
```

如果你从模型中忽略截距，每个分类值得列都会包括在设计矩阵的模型中：

```
In [54]: y, X = patsy.dmatrices('v2 ~ key1 + 0', data

In [55]: X
Out[55]:
DesignMatrix with shape (8, 2)
  key1[a]  key1[b]
        1        0
        1        0
        0        1
        0        1
        1        0
        0        1
        1        0
        0        1
  Terms:
    'key1' (columns 0:2)
```

使用C函数，数值列可以截取为分类量：

```
In [56]: y, X = patsy.dmatrices('v2 ~ C(key2)', data)

In [57]: X
Out[57]:
DesignMatrix with shape (8, 2)
  Intercept  C(key2)[T.1]
          1             0
          1             1
          1             0
          1             1
          1             0
          1             1
          1             0
          1             0
```

```
  Terms:
    'Intercept' (column 0)
    'C(key2)' (column 1)
```

当你在模型中使用多个分类名，事情就会变复杂，因为会包括key1:key2形式的相交部分，它可以用在方差（ANOVA）模型分析中：

```
In [58]: data['key2'] = data['key2'].map({0: 'zero', 

In [59]: data
Out[59]:
  key1  key2  v1   v2
0    a  zero   1 -1.0
1    a   one   2  0.0
2    b  zero   3  2.5
3    b   one   4 -0.5
4    a  zero   5  4.0
5    b   one   6 -1.2
6    a  zero   7  0.2
7    b  zero   8 -1.7

In [60]: y, X = patsy.dmatrices('v2 ~ key1 + key2', d

In [61]: X
Out[61]:
DesignMatrix with shape (8, 3)
  Intercept  key1[T.b]  key2[T.zero]
          1          0             1
          1          0             0
          1          1             1
          1          1             0
          1          0             1
          1          1             0
          1          0             1
          1          1             1
```

```
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
    'key2' (column 2)

In [62]: y, X = patsy.dmatrices('v2 ~ key1 + key2 + ke

In [63]: X
Out[63]:
DesignMatrix with shape (8, 4)
  Intercept  key1[T.b]  key2[T.zero]
key1[T.b]:key2[T.zero]
          1          0             1
          1          0             0
          1          1             1
          1          1             0
          1          0             1
          1          1             0
          1          0             1
          1          1             1
  Terms:
    'Intercept' (column 0)
    'key1' (column 1)
    'key2' (column 2)
    'key1:key2' (column 3)
```

Patsy提供转换分类数据的其它方法，包括以特定顺序转换。请参阅线上文档。

13.3 statsmodels介绍

statsmodels是Python进行拟合多种统计模型、进行统计试验和数据探索可视化的库。Statsmodels包含许多经典的统计方法，但没有贝叶斯方法和机器学习模型。

statsmodels包含的模型有：

- 线性模型，广义线性模型和健壮线性模型
- 线性混合效应模型
- 方差（ANOVA）方法分析
- 时间序列过程和状态空间模型
- 广义矩估计

下面，我会使用一些基本的statsmodels工具，探索Patsy公式和pandasDataFrame对象如何使用模型接口。

估计线性模型

statsmodels有多种线性回归模型，包括从基本（比如普通最小二乘）到复杂（比如迭代加权最小二乘法）的。

statsmodels的线性模型有两种不同的接口：基于数组，和基于公式。它们可以通过API模块引入：

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
```

为了展示它们的使用方法，我们从一些随机数据生成一个线性模型：

```
def dnorm(mean, variance, size=1):
    if isinstance(size, int):
        size = size,
    return mean + np.sqrt(variance) * np.random.randn

# For reproducibility
np.random.seed(12345)

N = 100
X = np.c_[dnorm(0, 0.4, size=N),
          dnorm(0, 0.6, size=N),
          dnorm(0, 0.2, size=N)]
eps = dnorm(0, 0.1, size=N)
beta = [0.1, 0.3, 0.5]
```

```
y = np.dot(X, beta) + eps
```

这里，我使用了“真实”模型和可知参数beta。此时，dnorm可用来生成正太分布数据，带有特定均值和方差。现在有：

```
In [66]: X[:5]
Out[66]:
array([[-0.1295, -1.2128,  0.5042],
       [ 0.3029, -0.4357, -0.2542],
       [-0.3285, -0.0253,  0.1384],
       [-0.3515, -0.7196, -0.2582],
       [ 1.2433, -0.3738, -0.5226]])

In [67]: y[:5]
Out[67]: array([ 0.4279, -0.6735, -0.0909, -0.4895,-0
```

像之前Patsy看到的，线性模型通常要拟合一个截距。sm.add_constant函数可以添加一个截距的列到现存的矩阵：

```
In [68]: X_model = sm.add_constant(X)

In [69]: X_model[:5]
Out[69]:
array([[ 1.    , -0.1295, -1.2128,  0.5042],
       [ 1.    ,  0.3029, -0.4357, -0.2542],
       [ 1.    , -0.3285, -0.0253,  0.1384],
       [ 1.    , -0.3515, -0.7196, -0.2582],
       [ 1.    ,  1.2433, -0.3738, -0.5226]])
```

sm.OLS类可以拟合一个普通最小二乘回归：

```
In [70]: model = sm.OLS(y, X)
```

这个模型的fit方法返回了一个回归结果对象，它包含估计的模型

参数和其它内容：

```
In [71]: results = model.fit()

In [72]: results.params
Out[72]: array([ 0.1783,  0.223 ,  0.501 ])
```

对结果使用summary方法可以打印模型的详细诊断结果：

```
In [73]: print(results.summary())
OLS Regression Results
=====================================================================
Dep. Variable:                      y   R-squared:
Model:                            OLS   Adj. R-square
Method:                 Least Squares   F-statistic:
Date:                Mon, 25 Sep 2017   Prob (F-stati
Time:                        14:06:15   Log-Likelihoo
No. Observations:                 100   AIC:
Df Residuals:                      97   BIC:
Df Model:                           3
Covariance Type:            nonrobust
=====================================================================
                 coef    std err          t      P>|t
---------------------------------------------------------------------
x1             0.1783      0.053      3.364      0.00
x2             0.2230      0.046      4.818      0.00
x3             0.5010      0.080      6.237      0.00
=====================================================================
Omnibus:                        4.662   Durbin-Watson
Prob(Omnibus):                  0.097   Jarque-Bera (
Skew:                           0.481   Prob(JB):
Kurtosis:                       3.243   Cond. No.
1.74
=====================================================================
Warnings:
[1] Standard Errors assume that the covariance matrix
specified.
```

这里的参数名为原始的名字x1, x2等等。假设所有的模型参数都在一个DataFrame中：

```
In [74]: data = pd.DataFrame(X, columns=['col0', 'col

In [75]: data['y'] = y

In [76]: data[:5]
Out[76]:
       col0      col1      col2         y
0 -0.129468 -1.212753  0.504225  0.427863
1  0.302910 -0.435742 -0.254180 -0.673480
2 -0.328522 -0.025302  0.138351 -0.090878
3 -0.351475 -0.719605 -0.258215 -0.489494
4  1.243269 -0.373799 -0.522629 -0.128941
```

现在，我们使用statsmodels的公式API和Patsy的公式字符串：

```
In [77]: results = smf.ols('y ~ col0 + col1 + col2', 

In [78]: results.params
Out[78]:
Intercept    0.033559
col0         0.176149
col1         0.224826
col2         0.514808
dtype: float64

In [79]: results.tvalues
Out[79]:
Intercept    0.952188
col0         3.319754
col1         4.850730
col2         6.303971
dtype: float64
```

观察下statsmodels是如何返回Series结果的，附带有DataFrame的列名。当使用公式和pandas对象时，我们不需要使用add_constant。

给出一个样本外数据，你可以根据估计的模型参数计算预测值：

```
In [80]: results.predict(data[:5])
Out[80]:
0   -0.002327
1   -0.141904
2    0.041226
3   -0.323070
4   -0.100535
dtype: float64
```

statsmodels的线性模型结果还有其它的分析、诊断和可视化工具。除了普通最小二乘模型，还有其它的线性模型。

估计时间序列过程

statsmodels的另一模型类是进行时间序列分析，包括自回归过程、卡尔曼滤波和其它态空间模型，和多元自回归模型。

用自回归结构和噪声来模拟一些时间序列数据：

```
init_x = 4

import random
values = [init_x, init_x]
N = 1000

b0 = 0.8
b1 = -0.4
noise = dnorm(0, 0.1, N)
for i in range(N):
    new_x = values[-1] * b0 + values[-2] * b1 + noise
```

```
    values.append(new_x)
```

这个数据有AR(2)结构（两个延迟），参数是0.8和-0.4。当你和AR模型，你可能不知道滞后项的个数，因此可以用较多的滞后量来拟合这个模型：

```
In [82]: MAXLAGS = 5

In [83]: model = sm.tsa.AR(values)

In [84]: results = model.fit(MAXLAGS)
```

结果中的估计参数首先是截距，其次是前两个参数的估计值：

```
In [85]: results.params
Out[85]: array([-0.0062,  0.7845, -0.4085, -0.0136,  (
```

更多的细节以及如何解释结果超出了本书的范围，可以通过statsmodels文档学习更多。

13.4 scikit-learn介绍

scikit-learn是一个广泛使用、用途多样的Python机器学习库。它包含多种标准监督和非监督机器学习方法和模型选择和评估、数据转换、数据加载和模型持久化工具。这些模型可以用于分类、聚合、预测和其它任务。

机器学习方面的学习和应用scikit-learn和TensorFlow解决实际问题的线上和纸质资料很多。本节中，我会简要介绍scikit-learn API的风格。

写作此书的时候，scikit-learn并没有和pandas深度结合，但是有些第三方包在开发中。尽管如此，pandas非常适合在模型拟合前处理数据集。

举个例子，我用一个Kaggle竞赛的经典数据集，关于泰坦尼克号乘客的生还率。我们用pandas加载测试和训练数据集：

```
In [86]: train = pd.read_csv('datasets/titanic/train.c

In [87]: test = pd.read_csv('datasets/titanic/test.csv

In [88]: train[:4]
Out[88]:
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
                                                Name
0                            Braund, Mr. Owen Harris
1  Cumings, Mrs. John Bradley (Florence Briggs Th...
2                             Heikkinen, Miss. Laina
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)
   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
3      0            113803  53.1000  C123        S
```

statsmodels和scikit-learn通常不能接收缺失数据，因此我们要查看列是否包含缺失值：

```
In [89]: train.isnull().sum()
Out[89]:
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
```

```
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64

In [90]: test.isnull().sum()
Out[90]:
PassengerId      0
Pclass           0
Name             0
Sex              0
Age             86
SibSp            0
Parch            0
Ticket           0
Fare             1
Cabin          327
Embarked         0
dtype: int64
```

在统计和机器学习的例子中，根据数据中的特征，一个典型的任务是预测乘客能否生还。模型现在训练数据集中拟合，然后用样本外测试数据集评估。

我想用年龄作为预测值，但是它包含缺失值。缺失数据补全的方法有多种，我用的是一种简单方法，用训练数据集的中位数补全两个表的空值：

```
In [91]: impute_value = train['Age'].median()

In [92]: train['Age'] = train['Age'].fillna(impute_va

In [93]: test['Age'] = test['Age'].fillna(impute_valu
```

现在我们需要指定模型。我增加了一个列IsFemale，作为“Sex”列的编码：

```
In [94]: train['IsFemale'] = (train['Sex'] == 'female

In [95]: test['IsFemale'] = (test['Sex'] == 'female')
```

然后，我们确定一些模型变量，并创建NumPy数组：

```
In [96]: predictors = ['Pclass', 'IsFemale', 'Age']

In [97]: X_train = train[predictors].values

In [98]: X_test = test[predictors].values

In [99]: y_train = train['Survived'].values

In [100]: X_train[:5]
Out[100]:
array([[  3.,   0.,  22.],
       [  1.,   1.,  38.],
       [  3.,   1.,  26.],
       [  1.,   1.,  35.],
       [  3.,   0.,  35.]])

In [101]: y_train[:5]
Out[101]: array([0, 1, 1, 1, 0])
```

我不能保证这是一个好模型，它的特征都符合。我们用scikit-learn的LogisticRegression模型，创建一个模型实例：

```
In [102]: from sklearn.linear_model import LogisticRe

In [103]: model = LogisticRegression()
```

与statsmodels类似，我们可以用模型的fit方法，将它拟合到训练数据：

```
In [104]: model.fit(X_train, y_train)
Out[104]:
LogisticRegression(C=1.0, class_weight=None, dual=Fal
          intercept_scaling=1, max_iter=100, multi_cl
          penalty='l2', random_state=None, solver='li
          verbose=0, warm_start=False)
```

现在，我们可以用model.predict，对测试数据进行预测：

```
In [105]: y_predict = model.predict(X_test)

In [106]: y_predict[:10]
Out[106]: array([0, 0, 0, 0, 1, 0, 1, 0, 1, 0])
```

如果你有测试数据集的真是值，你可以计算准确率或其它错误度量值：

```
(y_true == y_predict).mean()
```

在实际中，模型训练经常有许多额外的复杂因素。许多模型有可以调节的参数，有些方法（比如交叉验证）可以用来进行参数调节，避免对训练数据过拟合。这通常可以提高预测性或对新数据的健壮性。

交叉验证通过分割训练数据来模拟样本外预测。基于模型的精度得分（比如均方差），可以对模型参数进行网格搜索。有些模型，如logistic回归，有内置的交叉验证的估计类。例如，logisticregressioncv类可以用一个参数指定网格搜索对模型的正则化参数C的粒度：

```
In [107]: from sklearn.linear_model import LogisticRe

In [108]: model_cv = LogisticRegressionCV(10)
```

```
In [109]: model_cv.fit(X_train, y_train)
Out[109]:
LogisticRegressionCV(Cs=10, class_weight=None, cv=Non
           fit_intercept=True, intercept_scaling=1.0,
           multi_class='ovr', n_jobs=1, penalty='l2',
           refit=True, scoring=None, solver='lbfgs',
```

要手动进行交叉验证，你可以使用cross_val_score帮助函数，它可以处理数据分割。例如，要交叉验证我们的带有四个不重叠训练数据的模型，可以这样做：

```
In [110]: from sklearn.model_selection import cross_va

In [111]: model = LogisticRegression(C=10)

In [112]: scores = cross_val_score(model, X_train, y_

In [113]: scores
Out[113]: array([ 0.7723,  0.8027,  0.7703,  0.7883])
```

默认的评分指标取决于模型本身，但是可以明确指定一个评分狠话。交叉验证过的模型需要更长时间来训练，但会有更高的模型性能。

13.5 继续学习

我只是介绍了一些Python建模库的表面内容，现在有越来越多的框架用于各种统计和机器学习，它们都是用Python或Python用户界面实现的。

这本书的重点是数据规整，有其它的书是关注建模和数据科学工具的。其中优秀的有：

- Andreas Mueller and Sarah Guido (O’Reilly)的

《Introduction to Machine Learning with Python》

- Jake VanderPlas (O’Reilly)的 《Python Data Science Handbook》
- Joel Grus (O’Reilly) 的 《Data Science from Scratch: First Principles》
- Sebastian Raschka (Packt Publishing) 的《Python Machine Learning》
- Aurélien Géron (O’Reilly) 的《Hands-On Machine Learning with Scikit-Learn and TensorFlow》

虽然书是学习的好资源，但是随着底层开源软件的发展，书的内容会过时。最好是不断熟悉各种统计和机器学习框架的文档，学习最新的功能和API。

第14章 数据分析案例

本书正文的最后一章，我们来看一些真实世界的数据集。对于每个数据集，我们会用之前介绍的方法，从原始数据中提取有意义的内容。展示的方法适用于其它数据集，也包括你的。本章包含了一些各种各样的案例数据集，可以用来练习。

案例数据集可以在Github仓库找到，见第一章。

14.1 来自Bitly的USA.gov数据

2011年，URL缩短服务Bitly跟美国政府网站USA.gov合作，提供了一份从生成.gov或.mil短链接的用户那里收集来的匿名数据。在2011年，除实时数据之外，还可以下载文本文件形式的每小时快照。写作此书时（2017年），这项服务已经关闭，但我们保存一份数据用于本书的案例。

以每小时快照为例，文件中各行的格式为JSON（即JavaScript Object Notation，这是一种常用的Web数据格式）。例如，如果我们只读取某个文件中的第一行，那么所看到的结果应该是下面这样：

```
In [5]: path = 'datasets/bitly_usagov/example.txt'

In [6]: open(path).readline()
Out[6]: '{ "a": "Mozilla\\/5.0 (Windows NT 6.1; WOW64
(KHTML, like Gecko) Chrome\\/17.0.963.78 Safari\\/535
"tz": "America\\/New_York", "gr": "MA", "g": "A6qOVH"
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov",
"http:\\/\\/www.facebook.com\\/l\\/7AQEFzjSi\\/1.usa.
"http:\\/\\/www.ncbi.nlm.nih.gov\\/pubmed\\/22415991"
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.9
```

Python有内置或第三方模块可以将JSON字符串转换成Python字典对象。这里，我将使用json模块及其loads函数逐行加载已经下载好的数据文件：

```
import json
path = 'datasets/bitly_usagov/example.txt'
records = [json.loads(line) for line in open(path)]
```

现在，records对象就成为一组Python字典了：

```
In [18]: records[0]
Out[18]:
{'a': 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKi
Chrome/17.0.963.78 Safari/535.11',
 'al': 'en-US,en;q=0.8',
 'c': 'US',
 'cy': 'Danvers',
 'g': 'A6qOVH',
 'gr': 'MA',
 'h': 'wfLQtf',
 'hc': 1331822918,
 'hh': '1.usa.gov',
 'l': 'orofrog',
 'll': [42.576698, -70.954903],
 'nk': 1,
 'r': 'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/
 't': 1331923247,
 'tz': 'America/New_York',
 'u': 'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

用纯Python代码对时区进行计数

假设我们想要知道该数据集中最常出现的是哪个时区（即tz字段），得到答案的办法有很多。首先，我们用列表推导式取出一组时区：

```
In [12]: time_zones = [rec['tz'] for rec in records]
---------------------------------------------------------------
KeyError                                        Traceback (
```

```
<ipython-input-12-db4fbd348da9> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]
<ipython-input-12-db4fbd348da9> in <listcomp>(.0)
----> 1 time_zones = [rec['tz'] for rec in records]
KeyError: 'tz'
```

晕！原来并不是所有记录都有时区字段。这个好办，只需在列表推导式末尾加上一个if 'tz'in rec判断即可：

```
In [13]: time_zones = [rec['tz'] for rec in records i

In [14]: time_zones[:10]
Out[14]:
['America/New_York',
 'America/Denver',
 'America/New_York',
 'America/Sao_Paulo',
 'America/New_York',
 'America/New_York',
 'Europe/Warsaw',
 '',
 '',
 '']
```

只看前10个时区，我们发现有些是未知的（即空的）。虽然可以将它们过滤掉，但现在暂时先留着。接下来，为了对时区进行计数，这里介绍两个办法：一个较难（只使用标准Python库），另一个较简单（使用pandas）。计数的办法之一是在遍历时区的过程中将计数值保存在字典中：

```
def get_counts(sequence):
    counts = {}
    for x in sequence:
        if x in counts:
            counts[x] += 1
```

```
        else:
            counts[x] = 1
    return counts
```

如果使用Python标准库的更高级工具，那么你可能会将代码写得更简洁一些：

```
from collections import defaultdict

def get_counts2(sequence):
    counts = defaultdict(int) # values will initializ
    for x in sequence:
        counts[x] += 1
    return counts
```

我将逻辑写到函数中是为了获得更高的复用性。要用它对时区进行处理，只需将time_zones传入即可：

```
In [17]: counts = get_counts(time_zones)

In [18]: counts['America/New_York']
Out[18]: 1251

In [19]: len(time_zones)
Out[19]: 3440
```

如果想要得到前10位的时区及其计数值，我们需要用到一些有关字典的处理技巧：

```
def top_counts(count_dict, n=10):
    value_key_pairs = [(count, tz) for tz, count in c
    value_key_pairs.sort()
    return value_key_pairs[-n:]
```

然后有：

```
In [21]: top_counts(counts)
Out[21]:
[(33, 'America/Sao_Paulo'),
 (35, 'Europe/Madrid'),
(36, 'Pacific/Honolulu'),
 (37, 'Asia/Tokyo'),
 (74, 'Europe/London'),
 (191, 'America/Denver'),
 (382, 'America/Los_Angeles'),
 (400, 'America/Chicago'),
 (521, ''),
 (1251, 'America/New_York')]
```

如果你搜索Python的标准库，你能找到collections.Counter类，它可以使这项工作更简单：

```
In [22]: from collections import Counter

In [23]: counts = Counter(time_zones)

In [24]: counts.most_common(10)
Out[24]:
[('America/New_York', 1251),
 ('', 521),
 ('America/Chicago', 400),
 ('America/Los_Angeles', 382),
 ('America/Denver', 191),
 ('Europe/London', 74),
 ('Asia/Tokyo', 37),
 ('Pacific/Honolulu', 36),
 ('Europe/Madrid', 35),
 ('America/Sao_Paulo', 33)]
```

用pandas对时区进行计数

从原始记录的集合创建DateFrame，与将记录列表传递到pandas.DataFrame一样简单：

```
In [25]: import pandas as pd

In [26]: frame = pd.DataFrame(records)

In [27]: frame.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3560 entries, 0 to 3559
Data columns (total 18 columns):
_heartbeat_    120 non-null float64
a              3440 non-null object
al             3094 non-null object
c              2919 non-null object
cy             2919 non-null object
g              3440 non-null object
gr             2919 non-null object
h              3440 non-null object
hc             3440 non-null float64
hh             3440 non-null object
kw             93 non-null object
l              3440 non-null object
ll             2919 non-null object
nk             3440 non-null float64
r              3440 non-null object
t              3440 non-null float64
tz             3440 non-null object
u              3440 non-null object
dtypes: float64(4), object(14)
memory usage: 500.7+ KB

In [28]: frame['tz'][:10]
Out[28]:
0     America/New_York
1       America/Denver
2     America/New_York
3    America/Sao_Paulo
4     America/New_York
5     America/New_York
```

```
6         Europe/Warsaw
7
8
9
Name: tz, dtype: object
```

这里frame的输出形式是摘要视图（summary view），主要用于较大的DataFrame对象。我们然后可以对Series使用value_counts方法：

```
In [29]: tz_counts = frame['tz'].value_counts()

In [30]: tz_counts[:10]
Out[30]:
America/New_York       1251
                        521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Europe/London            74
Asia/Tokyo               37
Pacific/Honolulu         36
Europe/Madrid            35
America/Sao_Paulo        33
Name: tz, dtype: int64
```

我们可以用matplotlib可视化这个数据。为此，我们先给记录中未知或缺失的时区填上一个替代值。fillna函数可以替换缺失值（NA），而未知值（空字符串）则可以通过布尔型数组索引加以替换：

```
In [31]: clean_tz = frame['tz'].fillna('Missing')

In [32]: clean_tz[clean_tz == ''] = 'Unknown'

In [33]: tz_counts = clean_tz.value_counts()
```

```
In [34]: tz_counts[:10]
Out[34]:
America/New_York       1251
Unknown                 521
America/Chicago         400
America/Los_Angeles     382
America/Denver          191
Missing                 120
Europe/London            74
Asia/Tokyo               37
Pacific/Honolulu         36
Europe/Madrid            35
Name: tz, dtype: int64
```

此时，我们可以用seaborn包创建水平柱状图（结果见图14-1）：

```
In [36]: import seaborn as sns

In [37]: subset = tz_counts[:10]

In [38]: sns.barplot(y=subset.index, x=subset.values)
```

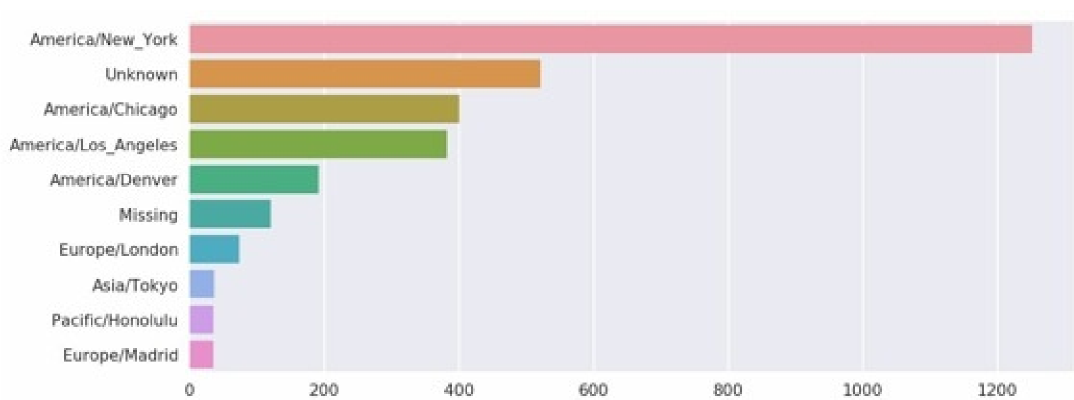


图14-1 usa.gov示例数据中最常出现的时区

a字段含有执行URL短缩操作的浏览器、设备、应用程序的相关信息：

```
In [39]: frame['a'][1]
Out[39]: 'GoogleMaps/RochesterNY'

In [40]: frame['a'][50]
Out[40]: 'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2)
Gecko/20100101 Firefox/10.0.2'

In [41]: frame['a'][51][:50]  # long line
Out[41]: 'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us
```

将这些"agent"字符串中的所有信息都解析出来是一件挺郁闷的工作。一种策略是将这种字符串的第一节（与浏览器大致对应）分离出来并得到另外一份用户行为摘要：

```
In [42]: results = pd.Series([x.split()[0] for x in f

In [43]: results[:5]
Out[43]:
0               Mozilla/5.0
1    GoogleMaps/RochesterNY
2               Mozilla/4.0
3               Mozilla/5.0
4               Mozilla/5.0
dtype: object

In [44]: results.value_counts()[:8]
Out[44]:
Mozilla/5.0                 2594
Mozilla/4.0                  601
GoogleMaps/RochesterNY       121
Opera/9.80                    34
TEST_INTERNET_AGENT           24
GoogleProducer                21
Mozilla/6.0                    5
BlackBerry8520/5.0.0.681       4
dtype: int64
```

现在，假设你想按Windows和非Windows用户对时区统计信息进行分解。为了简单起见，我们假定只要agent字符串中含有"Windows"就认为该用户为Windows用户。由于有的agent缺失，所以首先将它们从数据中移除：

```
In [45]: cframe = frame[frame.a.notnull()]
```

然后计算出各行是否含有Windows的值：

```
In [47]: cframe['os'] = np.where(cframe['a'].str.conta
   ....:                         'Windows', 'Not Windo

In [48]: cframe['os'][:5]
Out[48]:
0        Windows
1    Not Windows
2        Windows
3    Not Windows
4        Windows
Name: os, dtype: object
```

接下来就可以根据时区和新得到的操作系统列表对数据进行分组了：

```
In [49]: by_tz_os = cframe.groupby(['tz', 'os'])
```

分组计数，类似于value_counts函数，可以用size来计算。并利用unstack对计数结果进行重塑：

```
In [50]: agg_counts = by_tz_os.size().unstack().fillna

In [51]: agg_counts[:10]
Out[51]:
os                                Not Windows  Windows
tz
```

```
                                                   245.0    276.0
Africa/Cairo                                         0.0      3.0
Africa/Casablanca                                    0.0      1.0
Africa/Ceuta                                         0.0      2.0
Africa/Johannesburg                                  0.0      1.0
Africa/Lusaka                                        0.0      1.0
America/Anchorage                                    4.0      1.0
America/Argentina/Buenos_Aires                       1.0      0.0
America/Argentina/Cordoba                            0.0      1.0
America/Argentina/Mendoza                            0.0      1.0
```

最后，我们来选取最常出现的时区。为了达到这个目的，我根据agg_counts中的行数构造了一个间接索引数组：

```
# Use to sort in ascending order
In [52]: indexer = agg_counts.sum(1).argsort()

In [53]: indexer[:10]
Out[53]:
tz
                                  24
Africa/Cairo                      20
Africa/Casablanca                 21
Africa/Ceuta                      92
Africa/Johannesburg               87
Africa/Lusaka                     53
America/Anchorage                 54
America/Argentina/Buenos_Aires    57
America/Argentina/Cordoba         26
America/Argentina/Mendoza         55
dtype: int64
```

然后我通过take按照这个顺序截取了最后10行最大值：

```
In [54]: count_subset = agg_counts.take(indexer[-10:]
```

```
In [55]: count_subset
Out[55]:
os                    Not Windows  Windows
tz
America/Sao_Paulo            13.0     20.0
Europe/Madrid                16.0     19.0
Pacific/Honolulu              0.0     36.0
Asia/Tokyo                    2.0     35.0
Europe/London                43.0     31.0
America/Denver              132.0     59.0
America/Los_Angeles         130.0    252.0
America/Chicago             115.0    285.0
                            245.0    276.0
America/New_York            339.0    912.0
```

pandas有一个简便方法nlargest，可以做同样的工作：

```
In [56]: agg_counts.sum(1).nlargest(10)
Out[56]:
tz
America/New_York       1251.0
                        521.0
America/Chicago         400.0
America/Los_Angeles     382.0
America/Denver          191.0
Europe/London            74.0
Asia/Tokyo               37.0
Pacific/Honolulu         36.0
Europe/Madrid            35.0
America/Sao_Paulo        33.0
dtype: float64
```

然后，如这段代码所示，可以用柱状图表示。我传递一个额外参数到seaborn的barpolt函数，来画一个堆积条形图（见图14-2）：

```
# Rearrange the data for plotting
In [58]: count_subset = count_subset.stack()

In [59]: count_subset.name = 'total'

In [60]: count_subset = count_subset.reset_index()

In [61]: count_subset[:10]
Out[61]:
                  tz           os  total
0  America/Sao_Paulo  Not Windows   13.0
1  America/Sao_Paulo      Windows   20.0
2      Europe/Madrid  Not Windows   16.0
3      Europe/Madrid      Windows   19.0
4   Pacific/Honolulu  Not Windows    0.0
5   Pacific/Honolulu      Windows   36.0
6         Asia/Tokyo  Not Windows    2.0
7         Asia/Tokyo      Windows   35.0
8      Europe/London  Not Windows   43.0
9      Europe/London      Windows   31.0

In [62]: sns.barplot(x='total', y='tz', hue='os',  da
```

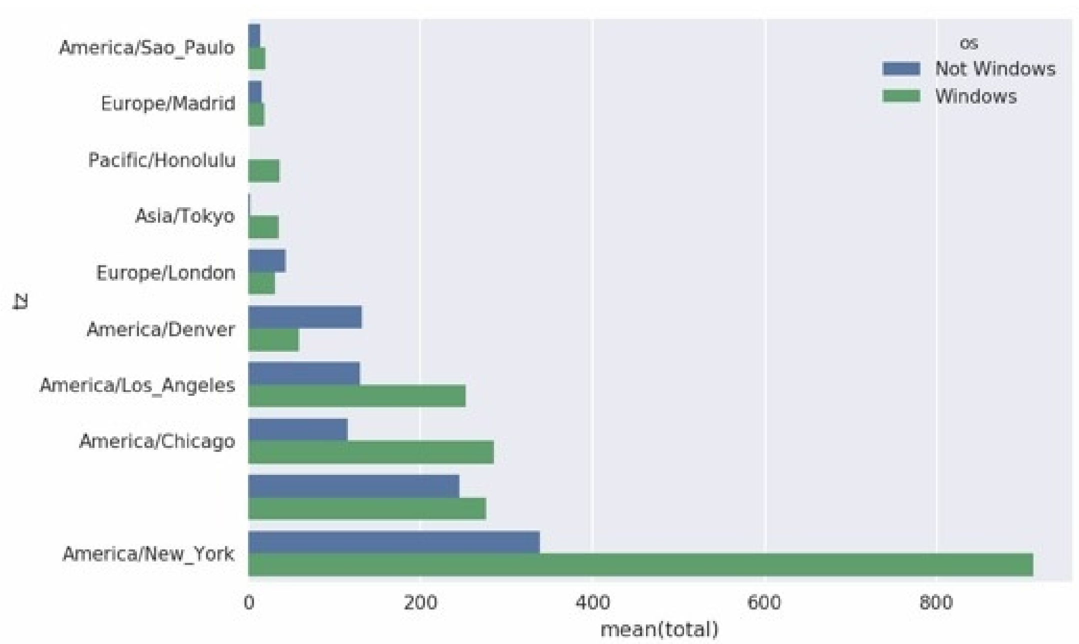


图14-2 最常出现时区的Windows和非Windows用户

这张图不容易看出Windows用户在小分组中的相对比例，因此标准化分组百分比之和为1：

```
def norm_total(group):
    group['normed_total'] = group.total / group.total

    return group

results = count_subset.groupby('tz').apply(norm_total
```

再次画图，见图14-3：

```
In [65]: sns.barplot(x='normed_total', y='tz', hue='o
```

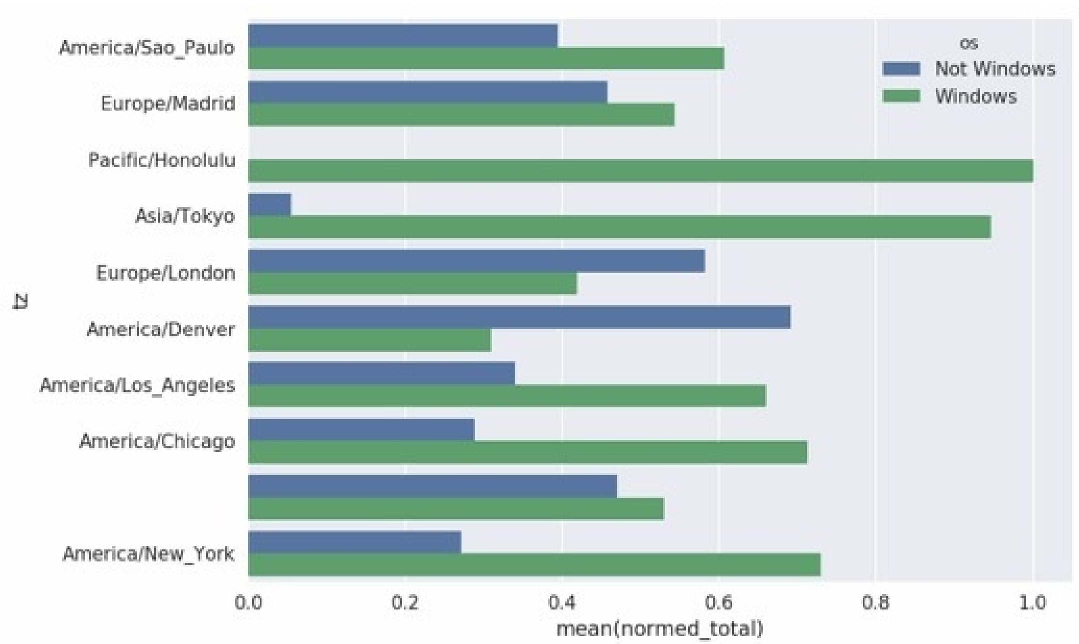


图14-3 最常出现时区的Windows和非Windows用户的百分比

我们还可以用groupby的transform方法，更高效的计算标准化的和：

```
In [66]: g = count_subset.groupby('tz')

In [67]: results2 = count_subset.total / g.total.tran
```

14.2 MovieLens 1M数据集

GroupLens Research（http://www.grouplens.org/node/73）采集了一组从20世纪90年末到21世纪初由MovieLens用户提供的电影评分数据。这些数据中包括电影评分、电影元数据（风格类型和年代）以及关于用户的人口统计学数据（年龄、邮编、性别和职业等）。基于机器学习算法的推荐系统一般都会对此类数据感兴趣。虽然我不会在本书中详细介绍机器学习技术，但我会告诉你如何对这种数据进行切片切块以满足实际需求。

MovieLens 1M数据集含有来自6000名用户对4000部电影的100万条评分数据。它分为三个表：评分、用户信息和电影信息。将该数据从zip文件中解压出来之后，可以通过pandas.read_table

将各个表分别读到一个pandas DataFrame对象中：

```
import pandas as pd

# Make display smaller
pd.options.display.max_rows = 10

unames = ['user_id', 'gender', 'age', 'occupation', ':
users = pd.read_table('datasets/movielens/users.dat',
                      header=None, names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp
ratings = pd.read_table('datasets/movielens/ratings.da
                        header=None, names=rnames)
mnames = ['movie_id', 'title', 'genres']
movies = pd.read_table('datasets/movielens/movies.dat
                       header=None, names=mnames)
```

利用Python的切片语法，通过查看每个DataFrame的前几行即可验证数据加载工作是否一切顺利：

```
In [69]: users[:5]
Out[69]:
   user_id gender  age  occupation    zip
0        1      F    1          10  48067
1        2      M   56          16  70072
2        3      M   25          15  55117
3        4      M   45           7  02460
4        5      M   25          20  55455

In [70]: ratings[:5]
Out[70]:
   user_id  movie_id  rating  timestamp
0        1      1193       5  978300760
1        1       661       3  978302109
2        1       914       3  978301968
```

```
3        1      3408        4  978300275
4        1      2355        5  978824291

In [71]: movies[:5]
Out[71]:
   movie_id                               title
0         1                    Toy Story (1995)   Ani
1         2                      Jumanji (1995)  Adve
2         3             Grumpier Old Men (1995)
3         4            Waiting to Exhale (1995)
4         5  Father of the Bride Part II (1995)

In [72]: ratings
Out[72]:
         user_id  movie_id  rating  timestamp
0              1      1193       5  978300760
1              1       661       3  978302109
2              1       914       3  978301968
3              1      3408       4  978300275
4              1      2355       5  978824291
...          ...       ...     ...        ...
1000204     6040      1091       1  956716541
1000205     6040      1094       5  956704887
1000206     6040       562       5  956704746
1000207     6040      1096       4  956715648
1000208     6040      1097       4  956715569
[1000209 rows x 4 columns]
```

注意，其中的年龄和职业是以编码形式给出的，它们的具体含义请参考该数据集的README文件。分析散布在三个表中的数据可不是一件轻松的事情。假设我们想要根据性别和年龄计算某部电影的平均得分，如果将所有数据都合并到一个表中的话问题就简单多了。我们先用pandas的merge函数将ratings跟users合并到一起，然后再将movies也合并进去。pandas会根据列名的重叠情况推断出哪些列是合并（或连接）键：

```
In [73]: data = pd.merge(pd.merge(ratings, users), mov

In [74]: data
Out[74]:
         user_id  movie_id  rating  timestamp gender
0              1      1193       5  978300760      F
1              2      1193       5  978298413      M
2             12      1193       4  978220179      M
3             15      1193       4  978199279      M
4             17      1193       5  978158471      M
...          ...       ...     ...        ...    ...
1000204     5949      2198       5  958846401      M
1000205     5675      2703       3  976029116      M
1000206     5780      2845       1  958153068      M
1000207     5851      3607       5  957756608      F
1000208     5938      2909       4  957273353      M
                                               title
0             One Flew Over the Cuckoo's Nest (1975)
1             One Flew Over the Cuckoo's Nest (1975)
2             One Flew Over the Cuckoo's Nest (1975)
3             One Flew Over the Cuckoo's Nest (1975)
4             One Flew Over the Cuckoo's Nest (1975)
...                                              ...
1000204                          Modulations (1998)
1000205                       Broken Vessels (1998)
1000206                           White Boys (1999)
1000207                     One Little Indian (1973)
1000208  Five Wives, Three Secretaries and Me (1998)
[1000209 rows x 10 columns]

In [75]: data.iloc[0]
Out[75]:
user_id                                                    1
movie_id                                                1193
rating                                                     5
timestamp                                          978300760
gender                                                     F
```

```
age                                                         1
occupation                                                 10
zip                                                     48067
title         One Flew Over the Cuckoo's Nest (1975)
genres                                                  Drama
Name: 0, dtype: object
```

为了按性别计算每部电影的平均得分，我们可以使用pivot_table方法：

```
In [76]: mean_ratings = data.pivot_table('rating', in
   ....:                                  columns='gen

In [77]: mean_ratings[:5]
Out[77]:
gender                                F         M
title
$1,000,000 Duck (1971)         3.375000  2.761905
'Night Mother (1986)           3.388889  3.352941
'Til There Was You (1997)      2.675676  2.733333
'burbs, The (1989)             2.793478  2.962085
...And Justice for All (1979)  3.828571  3.689024
```

该操作产生了另一个DataFrame，其内容为电影平均得分，行标为电影名称（索引），列标为性别。现在，我打算过滤掉评分数据不够250条的电影（随便选的一个数字）。为了达到这个目的，我先对title进行分组，然后利用size()得到一个含有各电影分组大小的Series对象：

```
In [78]: ratings_by_title = data.groupby('title').siz

In [79]: ratings_by_title[:10]
Out[79]:
title
$1,000,000 Duck (1971)                37
```

```
'Night Mother (1986)                   70
'Til There Was You (1997)              52
'burbs, The (1989)                    303
...And Justice for All (1979)         199
1-900 (1994)                            2
10 Things I Hate About You (1999)     700
101 Dalmatians (1961)                 565
101 Dalmatians (1996)                 364
12 Angry Men (1957)                   616
dtype: int64

In [80]: active_titles = ratings_by_title.index[ratin

In [81]: active_titles
Out[81]:
Index([''burbs, The (1989)', '10 Things I Hate About Y
       '101 Dalmatians (1961)', '101 Dalmatians (1996
       '13th Warrior, The (1999)', '2 Days in the Val
       '20,000 Leagues Under the Sea (1954)', '2001: A
       '2010 (1984)',
       ...
'X-Men (2000)', 'Year of Living Dangerously (1982)',
       'Yellow Submarine (1968)', 'You've Got Mail (1
       'Young Frankenstein (1974)', 'Young Guns (1988
       'Young Guns II (1990)', 'Young Sherlock Holmes
       'Zero Effect (1998)', 'eXistenZ (1999)'],
      dtype='object', name='title', length=1216)
```

标题索引中含有评分数据大于250条的电影名称，然后我们就可以据此从前面的mean_ratings中选取所需的行了：

```
# Select rows on the index
In [82]: mean_ratings = mean_ratings.loc[active_title

In [83]: mean_ratings
Out[83]:
```

```
gender                                        F         M
title
'burbs, The (1989)                     2.793478  2.962085
10 Things I Hate About You (1999)      3.646552  3.311966
101 Dalmatians (1961)                  3.791444  3.500000
101 Dalmatians (1996)                  3.240000  2.911215
12 Angry Men (1957)                    4.184397  4.328421
...                                         ...       ...
Young Guns (1988)                      3.371795  3.425620
Young Guns II (1990)                   2.934783  2.904025
Young Sherlock Holmes (1985)           3.514706  3.363344
Zero Effect (1998)                     3.864407  3.723140
eXistenZ (1999)                        3.098592  3.289086
[1216 rows x 2 columns]
```

为了了解女性观众最喜欢的电影，我们可以对F列降序排列：

```
In [85]: top_female_ratings = mean_ratings.sort_value

In [86]: top_female_ratings[:10]
Out[86]:
gender
title
Close Shave, A (1995)                                     4
Wrong Trousers, The (1993)                                4
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)             4
Wallace & Gromit: The Best of Aardman Animation...        4
Schindler's List (1993)                                   4
Shawshank Redemption, The (1994)                          4
Grand Day Out, A (1992)                                   4
To Kill a Mockingbird (1962)                              4
Creature Comforts (1990)                                  4
Usual Suspects, The (1995)                                4
```

计算评分分歧

假设我们想要找出男性和女性观众分歧最大的电影。一个办法是给mean_ratings加上一个用于存放平均得分之差的列，并对其进行排序：

```
In [87]: mean_ratings['diff'] = mean_ratings['M'] - me
```

按"diff"排序即可得到分歧最大且女性观众更喜欢的电影：

```
In [88]: sorted_by_diff = mean_ratings.sort_values(by=

In [89]: sorted_by_diff[:10]
Out[89]:
gender                                      F
title
Dirty Dancing (1987)                 3.790378  2.95
Jumpin' Jack Flash (1986)            3.254717  2.57
Grease (1978)                        3.975265  3.36
Little Women (1994)                  3.870588  3.32
Steel Magnolias (1989)               3.901734  3.36
Anastasia (1997)                     3.800000  3.28
Rocky Horror Picture Show, The (1975) 3.673016  3.16
Color Purple, The (1985)             4.158192  3.65
Age of Innocence, The (1993)         3.827068  3.33
Free Willy (1993)                    2.921348  2.43
```

对排序结果反序并取出前10行，得到的则是男性观众更喜欢的电影：

```
# Reverse order of rows, take first 10 rows
In [90]: sorted_by_diff[::-1][:10]
Out[90]:
gender                                       F
title
Good, The Bad and The Ugly, The (1966)  3.494949  4.2
Kentucky Fried Movie, The (1977)        2.878788  3.5
```

```
Dumb & Dumber (1994)                          2.697987  3.3
Longest Day, The (1962)                       3.411765  4.0
Cable Guy, The (1996)                         2.250000  2.8
Evil Dead II (Dead By Dawn) (1987)            3.297297  3.9
Hidden, The (1987)                            3.137931  3.7
Rocky III (1982)                              2.361702  2.9
Caddyshack (1980)                             3.396135  3.9
For a Few Dollars More (1965)                 3.409091  3.9
```

如果只是想要找出分歧最大的电影（不考虑性别因素），则可以计算得分数据的方差或标准差：

```
# Standard deviation of rating grouped by title
In [91]: rating_std_by_title = data.groupby('title')[

# Filter down to active_titles
In [92]: rating_std_by_title = rating_std_by_title.lo

# Order Series by value in descending order
In [93]: rating_std_by_title.sort_values(ascending=Fa
Out[93]:
title
Dumb & Dumber (1994)                           1.321333
Blair Witch Project, The (1999)                1.316368
Natural Born Killers (1994)                    1.307198
Tank Girl (1995)                               1.277695
Rocky Horror Picture Show, The (1975)          1.260177
Eyes Wide Shut (1999)                          1.259624
Evita (1996)                                   1.253631
Billy Madison (1995)                           1.249970
Fear and Loathing in Las Vegas (1998)          1.246408
Bicentennial Man (1999)                        1.245533
Name: rating, dtype: float64
```

可能你已经注意到了，电影分类是以竖线（|）分隔的字符串形式

给出的。如果想对电影分类进行分析的话，就需要先将其转换成更有用的形式才行。

14.3 1880-2010年间全美婴儿姓名

美国社会保障总署（SSA）提供了一份从1880年到现在的婴儿名字频率数据。Hadley Wickham（许多流行R包的作者）经常用这份数据来演示R的数据处理功能。

我们要做一些数据规整才能加载这个数据集，这么做就会产生一个如下的DataFrame：

```
In [4]: names.head(10)
Out[4]:
        name sex  births  year
0       Mary   F    7065  1880
1       Anna   F    2604  1880
2       Emma   F    2003  1880
3  Elizabeth   F    1939  1880
4     Minnie   F    1746  1880
5   Margaret   F    1578  1880
6        Ida   F    1472  1880
7      Alice   F    1414  1880
8     Bertha   F    1320  1880
9      Sarah   F    1288  1880
```

你可以用这个数据集做很多事，例如：

- 计算指定名字（可以是你自己的，也可以是别人的）的年度比例。
- 计算某个名字的相对排名。
- 计算各年度最流行的名字，以及增长或减少最快的名字。
- 分析名字趋势：元音、辅音、长度、总体多样性、拼写变化、首尾字母等。
- 分析外源性趋势：圣经中的名字、名人、人口结构变化等。

利用前面介绍过的那些工具，这些分析工作都能很轻松地完成，

我会讲解其中的一些。

到编写本书时为止，美国社会保障总署将该数据库按年度制成了多个数据文件，其中给出了每个性别/名字组合的出生总数。这些文件的原始档案可以在这里获取：http://www.ssa.gov/oact/babynames/limits.html。

如果你在阅读本书的时候这个页面已经不见了，也可以用搜索引擎找找。

下载"National data"文件names.zip，解压后的目录中含有一组文件（如yob1880.txt）。我用UNIX的head命令查看了其中一个文件的前10行（在Windows上，你可以用more命令，或直接在文本编辑器中打开）：

```
In [94]: !head -n 10 datasets/babynames/yob1880.txt
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

由于这是一个非常标准的以逗号隔开的格式，所以可以用pandas.read_csv将其加载到DataFrame中：

```
In [95]: import pandas as pd

In [96]: names1880 =
pd.read_csv('datasets/babynames/yob1880.txt',
   ....:                                 names=['name', 'sex'

In [97]: names1880
```

```
Out[97]:
            name sex  births
0           Mary   F    7065
1           Anna   F    2604
2           Emma   F    2003
3      Elizabeth   F    1939
4         Minnie   F    1746
...          ...  ..     ...
1995      Woodie   M       5
1996      Worthy   M       5
1997      Wright   M       5
1998        York   M       5
1999   Zachariah   M       5
[2000 rows x 3 columns]
```

这些文件中仅含有当年出现超过5次的名字。为了简单起见，我们可以用births列的sex分组小计表示该年度的births总计：

```
In [98]: names1880.groupby('sex').births.sum()
Out[98]:
sex
F     90993
M    110493
Name: births, dtype: int64
```

由于该数据集按年度被分隔成了多个文件，所以第一件事情就是要将所有数据都组装到一个DataFrame里面，并加上一个year字段。使用pandas.concat即可达到这个目的：

```
years = range(1880, 2011)

pieces = []
columns = ['name', 'sex', 'births']

for year in years:
    path = 'datasets/babynames/yob%d.txt' % year
```

```
    frame = pd.read_csv(path, names=columns)

    frame['year'] = year
    pieces.append(frame)

# Concatenate everything into a single DataFrame
names = pd.concat(pieces, ignore_index=True)
```

这里需要注意几件事情。第一，concat默认是按行将多个DataFrame组合到一起的；第二，必须指定ignore_index=True，因为我们不希望保留read_csv所返回的原始行号。现在我们得到了一个非常大的DataFrame，它含有全部的名字数据：

```
In [100]: names
Out[100]:
              name sex  births  year
0             Mary   F    7065  1880
1             Anna   F    2604  1880
2             Emma   F    2003  1880
3        Elizabeth   F    1939  1880
4           Minnie   F    1746  1880
...            ...  ..     ...   ...
1690779    Zymaire   M       5  2010
1690780     Zyonne   M       5  2010
1690781  Zyquarius   M       5  2010
1690782      Zyran   M       5  2010
1690783      Zzyzx   M       5  2010
[1690784 rows x 4 columns]
```

有了这些数据之后，我们就可以利用groupby或pivot_table在year和sex级别上对其进行聚合了，如图14-4所示：

```
In [101]: total_births = names.pivot_table('births', 
   .....:                                  columns='s
```

```
In [102]: total_births.tail()
Out[102]:
sex           F        M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382

In [103]: total_births.plot(title='Total births by se
```



图14-4 按性别和年度统计的总出生数

下面我们来插入一个prop列，用于存放指定名字的婴儿数相对于总出生数的比例。prop值为0.02表示每100名婴儿中有2名取了当前这个名字。因此，我们先按year和sex分组，然后再将新列加到各个分组上：

```
def add_prop(group):
    group['prop'] = group.births / group.births.sum()
    return group
names = names.groupby(['year', 'sex']).apply(add_prop
```

现在，完整的数据集就有了下面这些列：

```
In [105]: names
Out[105]:
              name sex  births  year      prop
0             Mary   F    7065  1880  0.077643
1             Anna   F    2604  1880  0.028618
2             Emma   F    2003  1880  0.022013
3        Elizabeth   F    1939  1880  0.021309
4          Minnie   F    1746  1880  0.019188
...            ...  ..     ...   ...       ...
1690779    Zymaire   M       5  2010  0.000003
1690780     Zyonne   M       5  2010  0.000003
1690781  Zyquarius   M       5  2010  0.000003
1690782      Zyran   M       5  2010  0.000003
1690783      Zzyzx   M       5  2010  0.000003
[1690784 rows x 5 columns]
```

在执行这样的分组处理时，一般都应该做一些有效性检查，比如验证所有分组的prop的总和是否为1：

```
In [106]: names.groupby(['year', 'sex']).prop.sum()
Out[106]:
year  sex
1880  F      1.0
      M      1.0
1881  F      1.0
      M      1.0
1882  F      1.0
            ...
2008  M      1.0
```

```
2009  F      1.0
      M      1.0
2010  F      1.0
      M      1.0
Name: prop, Length: 262, dtype: float64
```

工作完成。为了便于实现更进一步的分析，我需要取出该数据的一个子集：每对sex/year组合的前1000个名字。这又是一个分组操作：

```
def get_top1000(group):
    return group.sort_values(by='births', ascending=Fa
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
# Drop the group index, not needed
top1000.reset_index(inplace=True, drop=True)
```

如果你喜欢DIY的话，也可以这样：

```
pieces = []
for year, group in names.groupby(['year', 'sex']):
    pieces.append(group.sort_values(by='births', ascen
top1000 = pd.concat(pieces, ignore_index=True)
```

现在的结果数据集就小多了：

```
In [108]: top1000
Out[108]:
             name sex  births  year      prop
0            Mary   F    7065  1880  0.077643
1            Anna   F    2604  1880  0.028618
2            Emma   F    2003  1880  0.022013
3       Elizabeth   F    1939  1880  0.021309
4          Minnie   F    1746  1880  0.019188
...           ...  ..     ...   ...       ...
261872     Camilo   M     194  2010  0.000102
```

```
261873     Destin   M     194  2010  0.000102
261874     Jaquan   M     194  2010  0.000102
261875     Jaydan   M     194  2010  0.000102
261876     Maxton   M     193  2010  0.000102
[261877 rows x 5 columns]
```

接下来的数据分析工作就针对这个top1000数据集了。

分析命名趋势

有了完整的数据集和刚才生成的top1000数据集，我们就可以开始分析各种命名趋势了。首先将前1000个名字分为男女两个部分：

```
In [109]: boys = top1000[top1000.sex == 'M']

In [110]: girls = top1000[top1000.sex == 'F']
```

这是两个简单的时间序列，只需稍作整理即可绘制出相应的图表（比如每年叫做John和Mary的婴儿数）。我们先生成一张按year和name统计的总出生数透视表：

```
In [111]: total_births = top1000.pivot_table('births'
   .....:                                     columns=
   .....:                                     aggfunc=
```

现在，我们用DataFrame的plot方法绘制几个名字的曲线图（见图14-5）：

```
In [112]: total_births.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6868 entries, Aaden to Zuri
dtypes: float64(6868)
memory usage: 6.9 MB
```

```
In [113]: subset = total_births[['John', 'Harry', 'Ma

In [114]: subset.plot(subplots=True, figsize=(12, 10)
   .....:             title="Number of births per yea
```

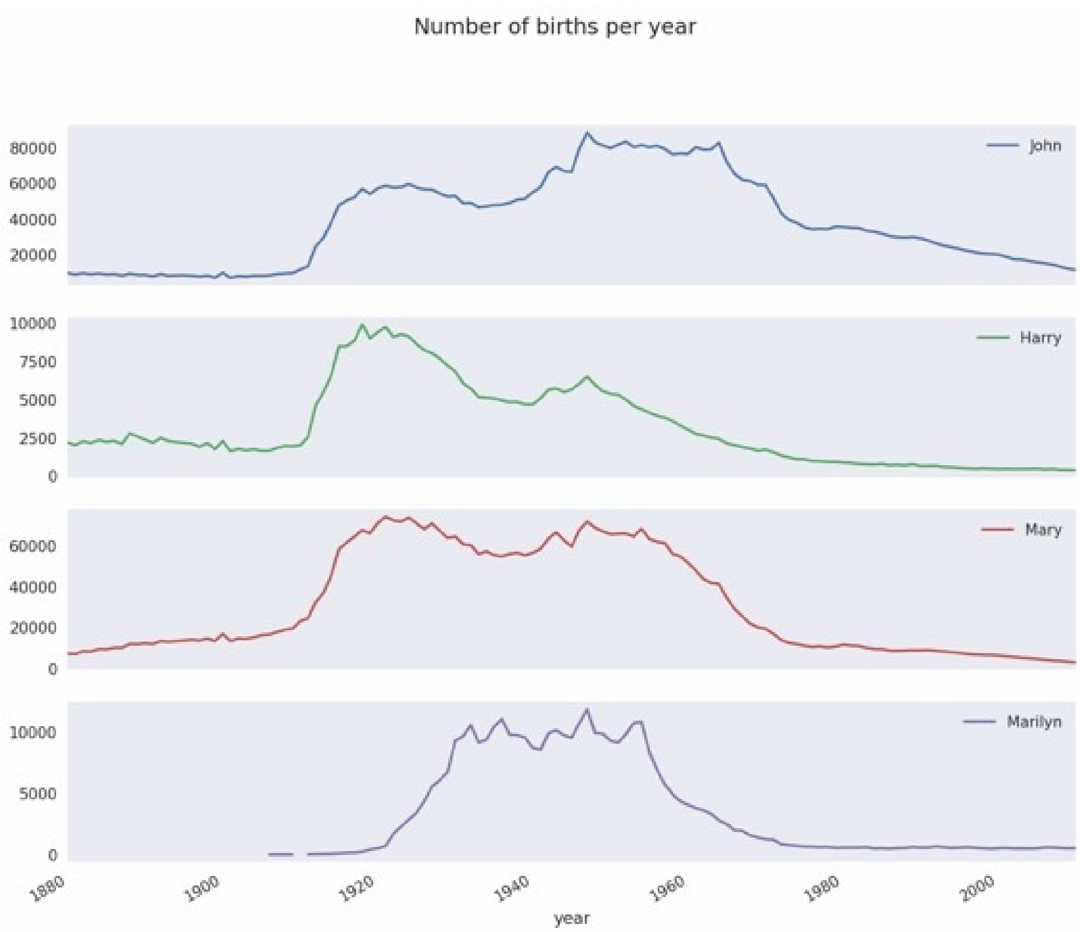


图14-5 几个男孩和女孩名字随时间变化的使用数量

从图中可以看出，这几个名字在美国人民的心目中已经风光不再了。但事实并非如此简单，我们在下一节中就能知道是怎么一回事了。

评估命名多样性的增长

一种解释是父母愿意给小孩起常见的名字越来越少。这个假设可以从数据中得到验证。一个办法是计算最流行的1000个名字所占的比例，我按year和sex进行聚合并绘图（见图14-6）：

```
In [116]: table = top1000.pivot_table('prop', index='
   .....:                             columns='sex', 

In [117]: table.plot(title='Sum of table1000.prop by 
   .....:            yticks=np.linspace(0, 1.2, 13), 
)
```

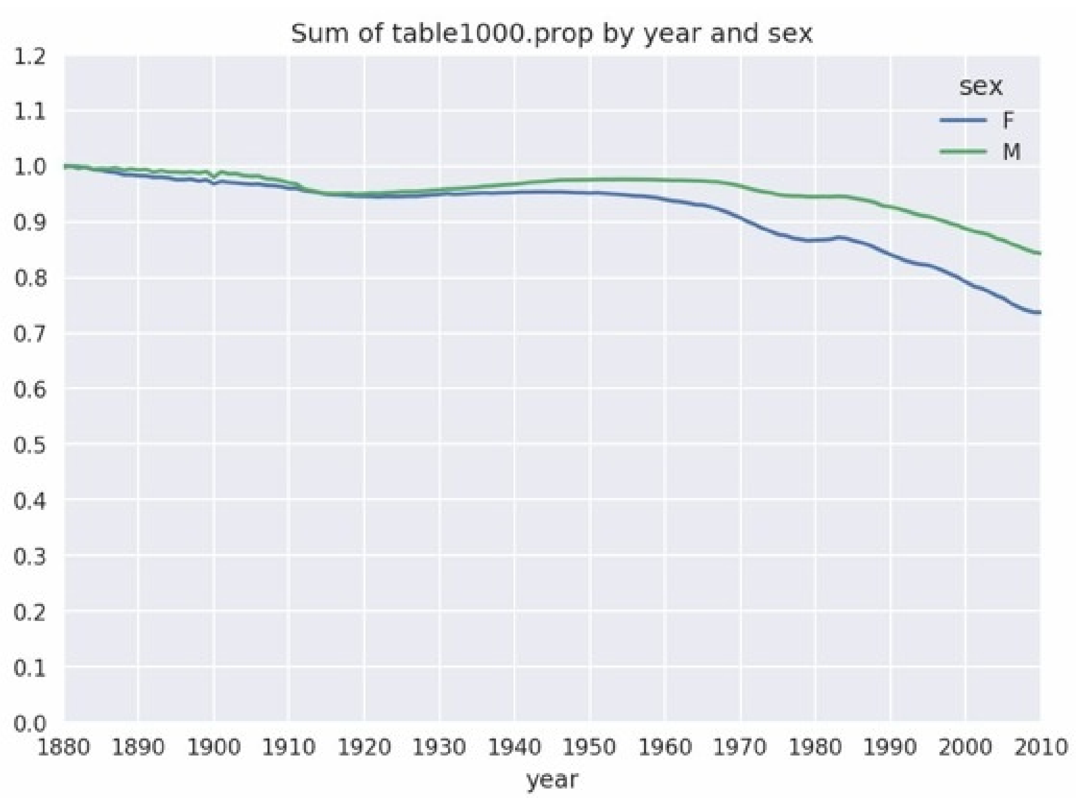


图14-6 分性别统计的前1000个名字在总出生人数中的比例

从图中可以看出，名字的多样性确实出现了增长（前1000项的比例降低）。另一个办法是计算占总出生人数前50%的不同名字的数量，这个数字不太好计算。我们只考虑2010年男孩的名字：

```
In [118]: df = boys[boys.year == 2010]

In [119]: df
Out[119]:
           name sex  births  year      prop
260877    Jacob   M   21875  2010  0.011523
```

```
260878     Ethan   M   17866  2010  0.009411
260879   Michael   M   17133  2010  0.009025
260880    Jayden   M   17030  2010  0.008971
260881   William   M   16870  2010  0.008887
...          ...  ..     ...   ...       ...
261872    Camilo   M     194  2010  0.000102
261873    Destin   M     194  2010  0.000102
261874    Jaquan   M     194  2010  0.000102
261875    Jaydan   M     194  2010  0.000102
261876    Maxton   M     193  2010  0.000102
[1000 rows x 5 columns]
```

在对prop降序排列之后，我们想知道前面多少个名字的人数加起来才够50%。虽然编写一个for循环确实也能达到目的，但NumPy有一种更聪明的矢量方式。先计算prop的累计和cumsum，然后再通过searchsorted方法找出0.5应该被插入在哪个位置才能保证不破坏顺序：

```
In [120]: prop_cumsum = df.sort_values(by='prop', asc

In [121]: prop_cumsum[:10]
Out[121]:
260877    0.011523
260878    0.020934
260879    0.029959
260880    0.038930
260881    0.047817
260882    0.056579
260883    0.065155
260884    0.073414
260885    0.081528
260886    0.089621
Name: prop, dtype: float64

In [122]: prop_cumsum.values.searchsorted(0.5)
Out[122]: 116
```

由于数组索引是从0开始的，因此我们要给这个结果加1，即最终结果为117。拿1900年的数据来做个比较，这个数字要小得多：

```
In [123]: df = boys[boys.year == 1900]

In [124]: in1900 = df.sort_values(by='prop', ascendin

In [125]: in1900.values.searchsorted(0.5) + 1
Out[125]: 25
```

现在就可以对所有year/sex组合执行这个计算了。按这两个字段进行groupby处理，然后用一个函数计算各分组的这个值：

```
def get_quantile_count(group, q=0.5):
    group = group.sort_values(by='prop', ascending=Fa
    return group.prop.cumsum().values.searchsorted(q)

diversity = top1000.groupby(['year', 'sex']).apply(ge
diversity = diversity.unstack('sex')
```

现在，diversity这个DataFrame拥有两个时间序列（每个性别各一个，按年度索引）。通过IPython，你可以查看其内容，还可以像之前那样绘制图表（如图14-7所示）：

```
In [128]: diversity.head()
Out[128]:
sex    F   M
year
1880  38  14
1881  38  14
1882  38  15
1883  39  15
1884  39  16

In [129]: diversity.plot(title="Number of popular nam
```

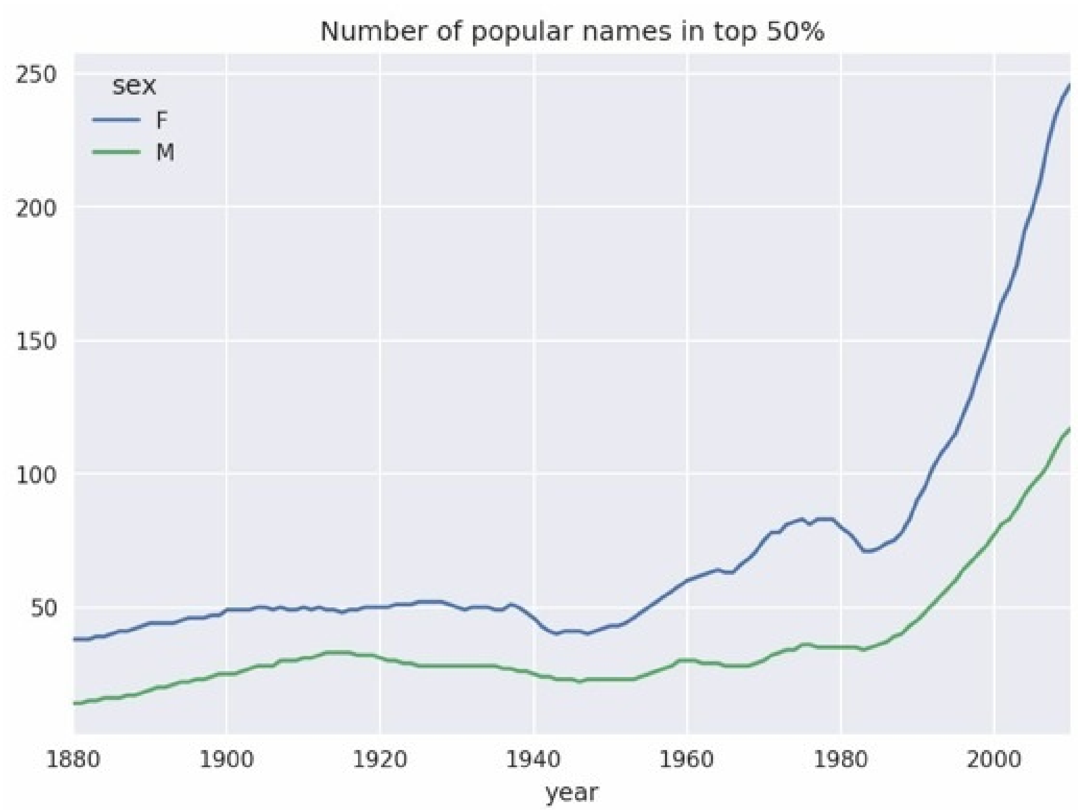


图14-7 按年度统计的密度表

从图中可以看出，女孩名字的多样性总是比男孩的高，而且还在变得越来越高。读者们可以自己分析一下具体是什么在驱动这个多样性（比如拼写形式的变化）。

“最后一个字母”的变革

2007年，一名婴儿姓名研究人员Laura Wattenberg在她自己的网站上指出（http://www.babynamewizard.com）：近百年来，男孩名字在最后一个字母上的分布发生了显著的变化。为了了解具体的情况，我首先将全部出生数据在年度、性别以及末字母上进行了聚合：

```
# extract last letter from name column
get_last_letter = lambda x: x[-1]
last_letters = names.name.map(get_last_letter)
last_letters.name = 'last_letter'
```

```
table = names.pivot_table('births', index=last_letter
                          columns=['sex', 'year'], ag
```

然后，我选出具有一定代表性的三年，并输出前面几行：

```
In [131]: subtable = table.reindex(columns=[1910, 196

In [132]: subtable.head()
Out[132]:
sex                  F                            M
year              1910      1960      2010     1910
last_letter
a             108376.0  691247.0  670605.0    977.0
b                  NaN     694.0     450.0    411.0
c                  5.0      49.0     946.0    482.0
d               6750.0    3729.0    2607.0  22111.0  2
e             133569.0  435013.0  313833.0  28655.0  1
```

接下来我们需要按总出生数对该表进行规范化处理，以便计算出各性别各末字母占总出生人数的比例：

```
In [133]: subtable.sum()
Out[133]:
sex  year
F    1910     396416.0
     1960    2022062.0
     2010    1759010.0
M    1910     194198.0
     1960    2132588.0
2010    1898382.0
dtype: float64

In [134]: letter_prop = subtable / subtable.sum()

In [135]: letter_prop
```

```
Out[135]:
sex                   F                            M
year               1910      1960      2010      1910
last_letter
a              0.273390  0.341853  0.381240  0.005031  (
b                   NaN  0.000343  0.000256  0.002116  (
c              0.000013  0.000024  0.000538  0.002482  (
d              0.017028  0.001844  0.001482  0.113858  (
e              0.336941  0.215133  0.178415  0.147556  (
...                 ...       ...       ...       ...
v                   NaN  0.000060  0.000117  0.000113
0.000037  0.001434
w              0.000020  0.000031  0.001182  0.006329  (
x              0.000015  0.000037  0.000727  0.003965  (
y              0.110972  0.152569  0.116828  0.077349  (
z              0.002439  0.000659  0.000704  0.000170  (
[26 rows x 6 columns]
```

有了这个字母比例数据之后，就可以生成一张各年度各性别的条形图了，如图14-8所示：

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0],
letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1],
                      legend=False)
```

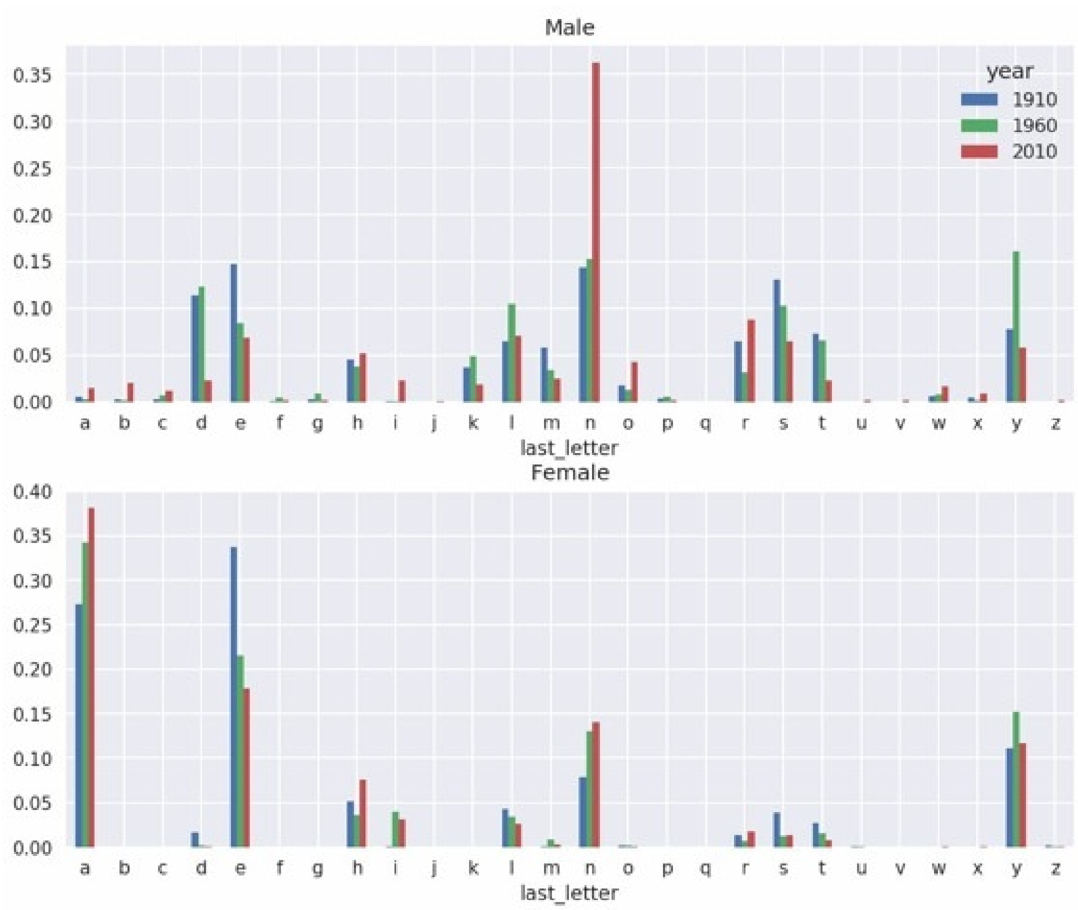


图14-8 男孩女孩名字中各个末字母的比例

可以看出，从20世纪60年代开始，以字母"n"结尾的男孩名字出现了显著的增长。回到之前创建的那个完整表，按年度和性别对其进行规范化处理，并在男孩名字中选取几个字母，最后进行转置以便将各个列做成一个时间序列：

```
In [138]: letter_prop = table / table.sum()

In [139]: dny_ts = letter_prop.loc[['d', 'n', 'y'], 'M

In [140]: dny_ts.head()
Out[140]:
last_letter          d         n         y
year
1880          0.083055  0.153213  0.075760
1881          0.083247  0.153214  0.077451
```

```
1882          0.085340  0.149560  0.077537
1883          0.084066  0.151646  0.079144
1884          0.086120  0.149915  0.080405
```

有了这个时间序列的DataFrame之后，就可以通过其plot方法绘制出一张趋势图了（如图14-9所示）：

```
In [143]: dny_ts.plot()
```

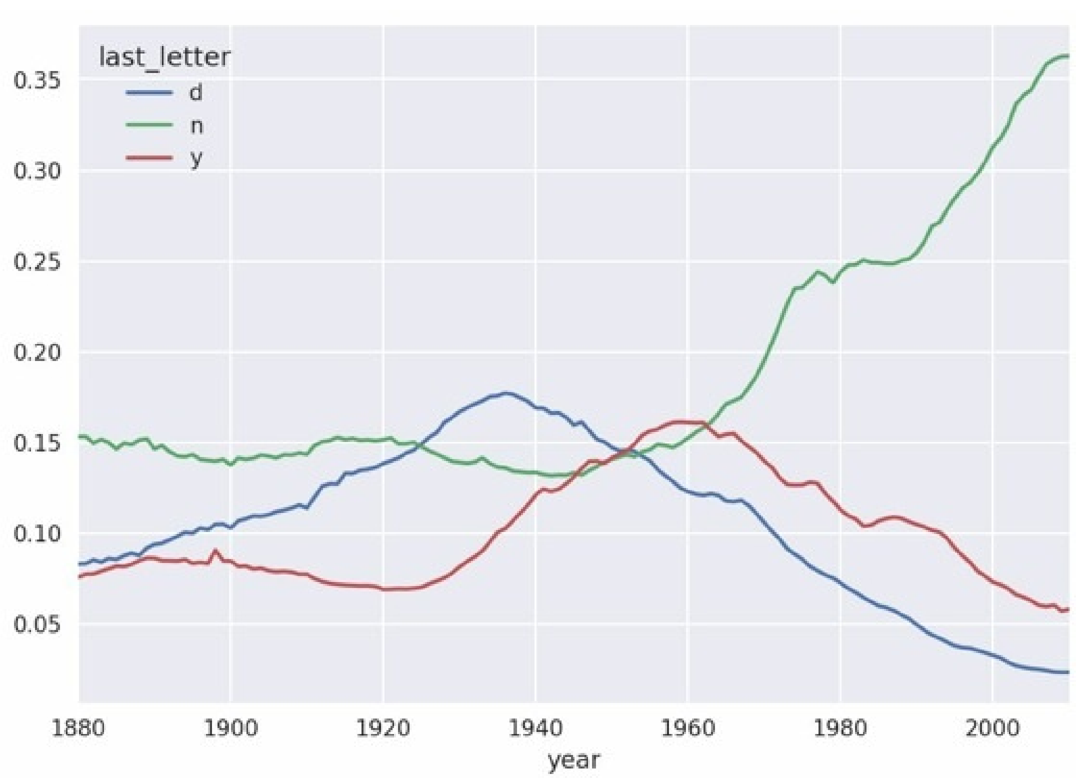


图14-9 各年出生的男孩中名字以d/n/y结尾的人数比例

变成女孩名字的男孩名字（以及相反的情况）

另一个有趣的趋势是，早年流行于男孩的名字近年来“变性了”，例如Lesley或Leslie。回到top1000数据集，找出其中以"lesl"开头的一组名字：

```
In [144]: all_names = pd.Series(top1000.name.unique()

In [145]: lesley_like = all_names[all_names.str.lower
```

```
In [146]: lesley_like
Out[146]:
632     Leslie
2294    Lesley
4262    Leslee
4728     Lesli
6103     Lesly
dtype: object
```

然后利用这个结果过滤其他的名字，并按名字分组计算出生数以查看相对频率：

```
In [147]: filtered = top1000[top1000.name.isin(lesley_

In [148]: filtered.groupby('name').births.sum()
Out[148]:
name
Leslee      1082
Lesley     35022
Lesli        929
Leslie    370429
Lesly      10067
Name: births, dtype: int64
```

接下来，我们按性别和年度进行聚合，并按年度进行规范化处理：

```
In [149]: table = filtered.pivot_table('births', inde
   .....:                               columns='sex',

In [150]: table = table.div(table.sum(1), axis=0)

In [151]: table.tail()
Out[151]:
sex     F   M
```

```
year
2006  1.0 NaN
2007  1.0 NaN
2008  1.0 NaN
2009  1.0 NaN
2010  1.0 NaN
```

最后，就可以轻松绘制一张分性别的年度曲线图了（如图2-10所示）：

```
In [153]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

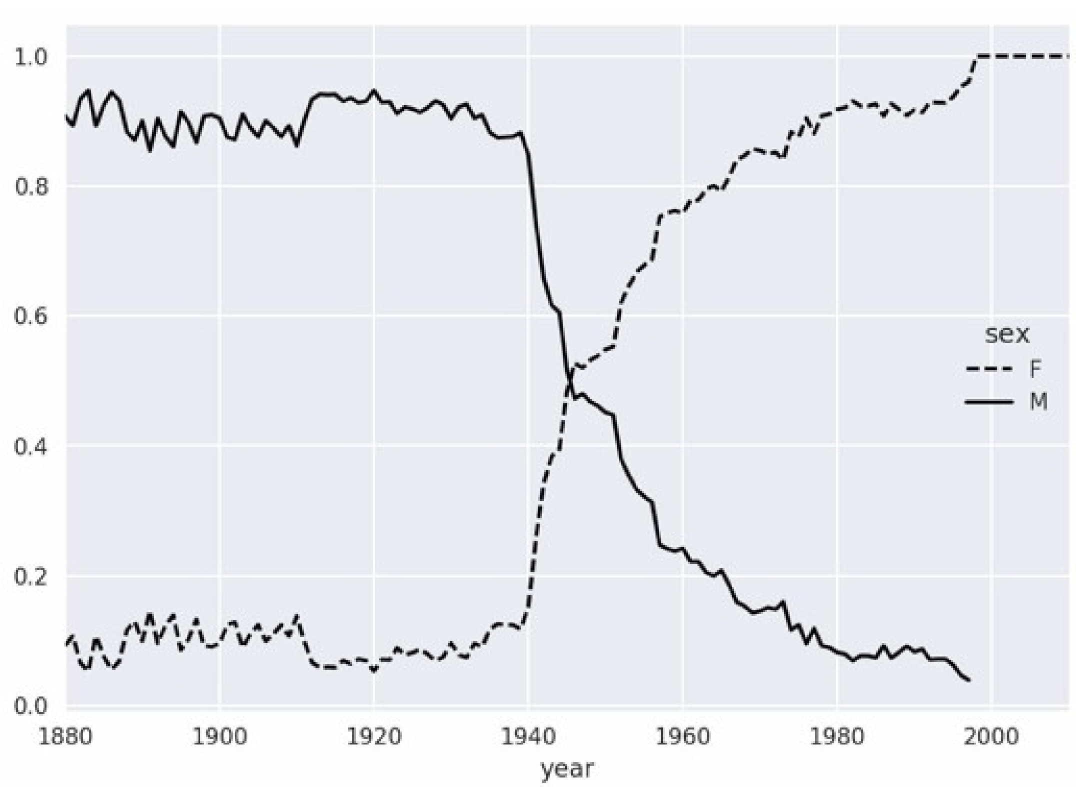


图14-10 各年度使用“Lesley型”名字的男女比例

14.4 USDA食品数据库

美国农业部（USDA）制作了一份有关食物营养信息的数据库。Ashley Williams制作了该数据的JSON版（http://ashleyw.co.uk/project/food-nutrient-database）。其中

的记录如下所示：

```
{
  "id": 21441,
  "description": "KENTUCKY FRIED CHICKEN, Fried Chick
Wing, meat and skin with breading",
  "tags": ["KFC"],
  "manufacturer": "Kentucky Fried Chicken",
"group": "Fast Foods",
  "portions": [
    {
      "amount": 1,
      "unit": "wing, with skin",
      "grams": 68.0
    },

    ...
  ],
  "nutrients": [
    {
      "value": 20.8,
      "units": "g",
      "description": "Protein",
      "group": "Composition"
    },

    ...
  ]
}
```

每种食物都带有若干标识性属性以及两个有关营养成分和分量的列表。这种形式的数据不是很适合分析工作，因此我们需要做一些规整化以使其具有更好用的形式。

从上面列举的那个网址下载并解压数据之后，你可以用任何喜欢的JSON库将其加载到Python中。我用的是Python内置的json模

块：

```
In [154]: import json

In [155]: db = json.load(open('datasets/usda_food/data

In [156]: len(db)
Out[156]: 6636
```

db中的每个条目都是一个含有某种食物全部数据的字典。nutrients字段是一个字典列表，其中的每个字典对应一种营养成分：

```
In [157]: db[0].keys()
Out[157]: dict_keys(['id', 'description', 'tags', 'ma
ons', 'nutrients'])

In [158]: db[0]['nutrients'][0]
Out[158]:
{'description': 'Protein',
 'group': 'Composition',
 'units': 'g',
 'value': 25.18}

In [159]: nutrients = pd.DataFrame(db[0]['nutrients']

In [160]: nutrients[:7]
Out[160]:
                   description        group units
0                      Protein  Composition     g
1            Total lipid (fat)  Composition     g
2  Carbohydrate, by difference  Composition     g
3                          Ash        Other     g
4                       Energy       Energy  kcal   3
5                        Water  Composition     g
6                       Energy       Energy    kJ  15
```

在将字典列表转换为DataFrame时，可以只抽取其中的一部分字段。这里，我们将取出食物的名称、分类、编号以及制造商等信息：

```
In [161]: info_keys = ['description', 'group', 'id',

In [162]: info = pd.DataFrame(db, columns=info_keys)

In [163]: info[:5]
Out[163]:
                          description
0                     Cheese, caraway  Dairy and Egg 
1                     Cheese, cheddar  Dairy and Egg 
2                        Cheese, edam  Dairy and Egg 
3                        Cheese, feta  Dairy and Egg 
4  Cheese, mozzarella, part skim milk  Dairy and Egg 
  manufacturer
0
1
2
3
4

In [164]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
description     6636 non-null object
group           6636 non-null object
id              6636 non-null int64
manufacturer    5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB
```

通过value_counts，你可以查看食物类别的分布情况：

```
In [165]: pd.value_counts(info.group)[:10]
Out[165]:
Vegetables and Vegetable Products    812
Beef Products                        618
Baked Products                       496
Breakfast Cereals                    403
Fast Foods                           365
Legumes and Legume Products          365
Lamb, Veal, and Game Products        345
Sweets                               341
Pork Products                        328
Fruits and Fruit Juices              328
Name: group, dtype: int64
```

现在，为了对全部营养数据做一些分析，最简单的办法是将所有食物的营养成分整合到一个大表中。我们分几个步骤来实现该目的。首先，将各食物的营养成分列表转换为一个DataFrame，并添加一个表示编号的列，然后将该DataFrame添加到一个列表中。最后通过concat将这些东西连接起来就可以了：

顺利的话，nutrients的结果是：

```
In [167]: nutrients
Out[167]:
                               description        gro
0                                  Protein  Compositi
1                        Total lipid (fat)  Compositi
2              Carbohydrate, by difference  Compositi
3                                      Ash       Oth
4                                   Energy      Ener
...                                    ...         .
...       ...     ...
389350                Vitamin B-12, added     Vitami
389351                         Cholesterol       Oth
389352         Fatty acids, total saturated       Oth
389353  Fatty acids, total monounsaturated       Oth
389354  Fatty acids, total polyunsaturated       Oth
```

```
[389355 rows x 5 columns]
```

我发现这个DataFrame中无论如何都会有一些重复项，所以直接丢弃就可以了：

```
In [168]: nutrients.duplicated().sum()  # number of d
Out[168]: 14179

In [169]: nutrients = nutrients.drop_duplicates()
```

由于两个DataFrame对象中都有"group"和"description"，所以为了明确到底谁是谁，我们需要对它们进行重命名：

```
In [170]: col_mapping = {'description' : 'food',
   .....:                'group'       : 'fgroup'}

In [171]: info = info.rename(columns=col_mapping, cop

In [172]: info.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6636 entries, 0 to 6635
Data columns (total 4 columns):
food            6636 non-null object
fgroup          6636 non-null object
id              6636 non-null int64
manufacturer    5195 non-null object
dtypes: int64(1), object(3)
memory usage: 207.5+ KB

In [173]: col_mapping = {'description' : 'nutrient',
   .....:                'group' : 'nutgroup'}
In [174]: nutrients = nutrients.rename(columns=col_ma

In [175]: nutrients
Out[175]:
                                  nutrient     nutgro
```

```
0                                   Protein  Compositi
1                         Total lipid (fat)  Compositi
2                 Carbohydrate, by difference  Compositi
3                                       Ash       Oth
4                                    Energy      Ener
...                                     ...         .
389350                  Vitamin B-12, added     Vitami
389351                          Cholesterol       Oth
389352          Fatty acids, total saturated       Oth
389353  Fatty acids, total monounsaturated       Oth
389354  Fatty acids, total polyunsaturated       Oth
[375176 rows x 5 columns]
```

做完这些，就可以将info跟nutrients合并起来：

```
In [176]: ndata = pd.merge(nutrients, info, on='id', 

In [177]: ndata.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns (total 8 columns):
nutrient        375176 non-null object
nutgroup        375176 non-null object
units           375176 non-null object
value           375176 non-null float64
id              375176 non-null int64
food            375176 non-null object
fgroup          375176 non-null object
manufacturer    293054 non-null object
dtypes: float64(1), int64(1), object(6)
memory usage: 25.8+ MB

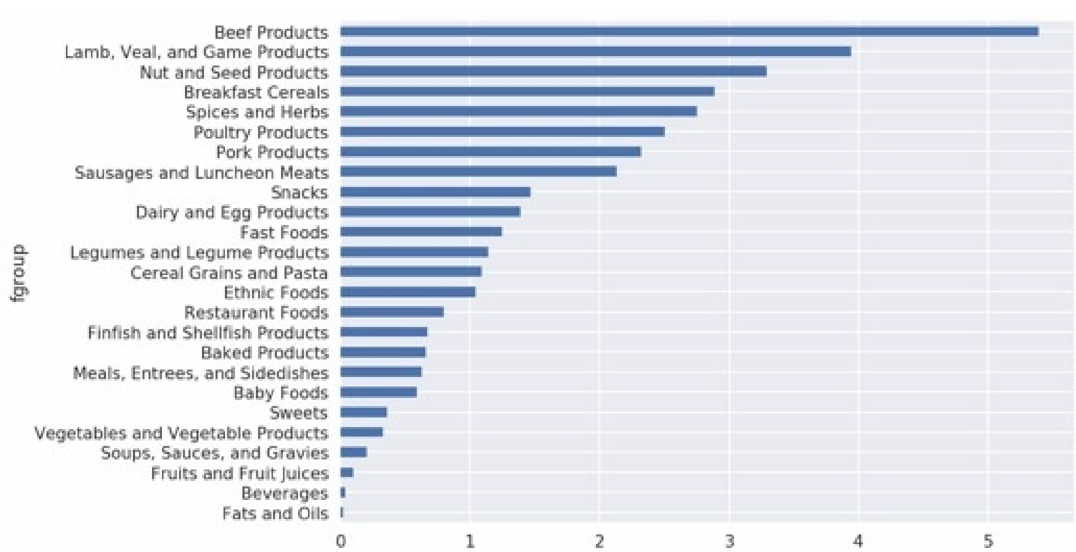
In [178]: ndata.iloc[30000]
Out[178]:
nutrient                                        Glycin
nutgroup                                    Amino Acid
```

```
units                                                              (
value                                                           0.04
id                                                              615
food          Soup, tomato bisque, canned, condense
fgroup                            Soups, Sauces, and Gravie
manufacturer
Name: 30000, dtype: object
```

我们现在可以根据食物分类和营养类型画出一张中位值图（如图14-11所示）：

```
In [180]: result = ndata.groupby(['nutrient', 'fgroup

In [181]: result['Zinc, Zn'].sort_values().plot(kind=
```



图片14-11 根据营养分类得出的锌中位值

只要稍微动一动脑子，就可以发现各营养成分最为丰富的食物是什么了：

```
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])

get_maximum = lambda x: x.loc[x.value.idxmax()]
get_minimum = lambda x: x.loc[x.value.idxmin()]
```

```
max_foods = by_nutrient.apply(get_maximum)[['value',

# make the food a little smaller
max_foods.food = max_foods.food.str[:50]
```

由于得到的DataFrame很大，所以不方便在书里面全部打印出来。这里只给出"Amino Acids"营养分组：

```
In [183]: max_foods.loc['Amino Acids']['food']
Out[183]:
nutrient
Alanine                          Gelatins, dry powder
Arginine                              Seeds, sesame f
Aspartic acid                                  Soy pr
Cystine              Seeds, cottonseed flour, low fa
Glutamic acid                                  Soy pr
                                          ...
Serine           Soy protein isolate, PROTEIN TECHNOL
Threonine        Soy protein isolate, PROTEIN TECHNOL
Tryptophan        Sea lion, Steller, meat with fat (A
Tyrosine         Soy protein isolate, PROTEIN TECHNOL
Valine           Soy protein isolate, PROTEIN TECHNOL
Name: food, Length: 19, dtype: object
```

14.5 2012联邦选举委员会数据库

美国联邦选举委员会发布了有关政治竞选赞助方面的数据。其中包括赞助者的姓名、职业、雇主、地址以及出资额等信息。我们对2012年美国总统大选的数据集比较感兴趣（http://www.fec.gov/disclosurep/PDownload.do）。我在2012年6月下载的数据集是一个150MB的CSV文件（P00000001-ALL.csv），我们先用pandas.read_csv将其加载进来：

```
In [184]: fec = pd.read_csv('datasets/fec/P00000001-A
```

```
In [185]: fec.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1001731 entries, 0 to 1001730
Data columns (total 16 columns):
cmte_id              1001731 non-null object
cand_id              1001731 non-null object
cand_nm              1001731 non-null object
contbr_nm            1001731 non-null object
contbr_city          1001712 non-null object
contbr_st            1001727 non-null object
contbr_zip           1001620 non-null object
contbr_employer      988002 non-null object
contbr_occupation    993301 non-null object
contb_receipt_amt    1001731 non-null float64
contb_receipt_dt     1001731 non-null object
receipt_desc         14166 non-null object
memo_cd              92482 non-null object
memo_text            97770 non-null object
form_tp              1001731 non-null object
file_num             1001731 non-null int64
dtypes: float64(1), int64(1), object(14)
memory usage: 122.3+ MB
```

该DataFrame中的记录如下所示：

```
In [186]: fec.iloc[123456]
Out[186]:
cmte_id             C00431445
cand_id             P80003338
cand_nm         Obama, Barack
contbr_nm         ELLMAN, IRA
contbr_city             TEMPE
                     ...
receipt_desc              NaN
memo_cd                   NaN
```

```
memo_text                          NaN
form_tp                          SA17A
file_num                        772372
Name: 123456, Length: 16, dtype: object
```

你可能已经想出了许多办法从这些竞选赞助数据中抽取有关赞助人和赞助模式的统计信息。我将在接下来的内容中介绍几种不同的分析工作（运用到目前为止已经学到的方法）。

不难看出，该数据中没有党派信息，因此最好把它加进去。通过unique，你可以获取全部的候选人名单：

```
In [187]: unique_cands = fec.cand_nm.unique()

In [188]: unique_cands
Out[188]:
array(['Bachmann, Michelle', 'Romney, Mitt', 'Obama, 
       "Roemer, Charles E. 'Buddy' III", 'Pawlenty, T
       'Johnson, Gary Earl', 'Paul, Ron', 'Santorum, 
       'Gingrich, Newt', 'McCotter, Thaddeus G', 'Hun
       'Perry, Rick'], dtype=object)

In [189]: unique_cands[2]
Out[189]: 'Obama, Barack'
```

指明党派信息的方法之一是使用字典：

```
parties = {'Bachmann, Michelle': 'Republican',
           'Cain, Herman': 'Republican',
           'Gingrich, Newt': 'Republican',
           'Huntsman, Jon': 'Republican',
           'Johnson, Gary Earl': 'Republican',
           'McCotter, Thaddeus G': 'Republican',
           'Obama, Barack': 'Democrat',
           'Paul, Ron': 'Republican',
           'Pawlenty, Timothy': 'Republican',
```

```
            'Perry, Rick': 'Republican',
            "Roemer, Charles E. 'Buddy' III": 'Republi
            'Romney, Mitt': 'Republican',
            'Santorum, Rick': 'Republican'}
```

现在，通过这个映射以及Series对象的map方法，你可以根据候选人姓名得到一组党派信息：

```
In [191]: fec.cand_nm[123456:123461]
Out[191]:
123456    Obama, Barack
123457    Obama, Barack
123458    Obama, Barack
123459    Obama, Barack
123460    Obama, Barack
Name: cand_nm, dtype: object

In [192]: fec.cand_nm[123456:123461].map(parties)
Out[192]:
123456    Democrat
123457    Democrat
123458    Democrat
123459    Democrat
123460    Democrat
Name: cand_nm, dtype: object

# Add it as a column
In [193]: fec['party'] = fec.cand_nm.map(parties)

In [194]: fec['party'].value_counts()
Out[194]:
Democrat      593746
Republican    407985
Name: party, dtype: int64
```

这里有两个需要注意的地方。第一，该数据既包括赞助也包括退

款（负的出资额）：

```
In [195]: (fec.contb_receipt_amt > 0).value_counts()
Out[195]:
True     991475
False     10256
Name: contb_receipt_amt, dtype: int64
```

为了简化分析过程，我限定该数据集只能有正的出资额：

```
In [196]: fec = fec[fec.contb_receipt_amt > 0]
```

由于Barack Obama和Mitt Romney是最主要的两名候选人，所以我还专门准备了一个子集，只包含针对他们两人的竞选活动的赞助信息：

```
In [197]: fec_mrbo = fec[fec.cand_nm.isin(['Obama, Ba
```

根据职业和雇主统计赞助信息

基于职业的赞助信息统计是另一种经常被研究的统计任务。例如，律师们更倾向于资助民主党，而企业主则更倾向于资助共和党。你可以不相信我，自己看那些数据就知道了。首先，根据职业计算出资总额，这很简单：

```
In [198]: fec.contbr_occupation.value_counts()[:10]
Out[198]:
RETIRED                                   233990
INFORMATION REQUESTED                      35107
ATTORNEY                                   34286
HOMEMAKER                                  29931
PHYSICIAN                                  23432
INFORMATION REQUESTED PER BEST EFFORTS     21138
ENGINEER                                   14334
TEACHER                                    13990
CONSULTANT                                 13273
```

```
PROFESSOR                                             12555
Name: contbr_occupation, dtype: int64
```

不难看出，许多职业都涉及相同的基本工作类型，或者同一样东西有多种变体。下面的代码片段可以清理一些这样的数据（将一个职业信息映射到另一个）。注意，这里巧妙地利用了dict.get，它允许没有映射关系的职业也能“通过”：

```
occ_mapping = {
   'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PR
   'INFORMATION REQUESTED' : 'NOT PROVIDED',
   'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROV
   'C.E.O.': 'CEO'
}

# If no mapping provided, return x
f = lambda x: occ_mapping.get(x, x)
fec.contbr_occupation = fec.contbr_occupation.map(f)
```

我对雇主信息也进行了同样的处理：

```
emp_mapping = {
   'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PR
   'INFORMATION REQUESTED' : 'NOT PROVIDED',
   'SELF' : 'SELF-EMPLOYED',
   'SELF EMPLOYED' : 'SELF-EMPLOYED',
}

# If no mapping provided, return x
f = lambda x: emp_mapping.get(x, x)
fec.contbr_employer = fec.contbr_employer.map(f)
```

现在，你可以通过pivot_table根据党派和职业对数据进行聚合，然后过滤掉总出资额不足200万美元的数据：

```
In [201]: by_occupation = fec.pivot_table('contb_rece
   .....:                                 index='contb
   .....:                                 columns='pa

In [202]: over_2mm = by_occupation[by_occupation.sum(

In [203]: over_2mm
Out[203]:
party                 Democrat    Republican
contbr_occupation
ATTORNEY           11141982.97  7.477194e+06
CEO                 2074974.79  4.211041e+06
CONSULTANT          2459912.71  2.544725e+06
ENGINEER             951525.55  1.818374e+06
EXECUTIVE           1355161.05  4.138850e+06
...                        ...           ...
PRESIDENT           1878509.95  4.720924e+06
PROFESSOR           2165071.08  2.967027e+05
REAL ESTATE          528902.09  1.625902e+06
RETIRED            25305116.38  2.356124e+07
SELF-EMPLOYED        672393.40  1.640253e+06
[17 rows x 2 columns]
```

把这些数据做成柱状图看起来会更加清楚（'barh'表示水平柱状图，如图14-12所示）：

```
In [205]: over_2mm.plot(kind='barh')
```

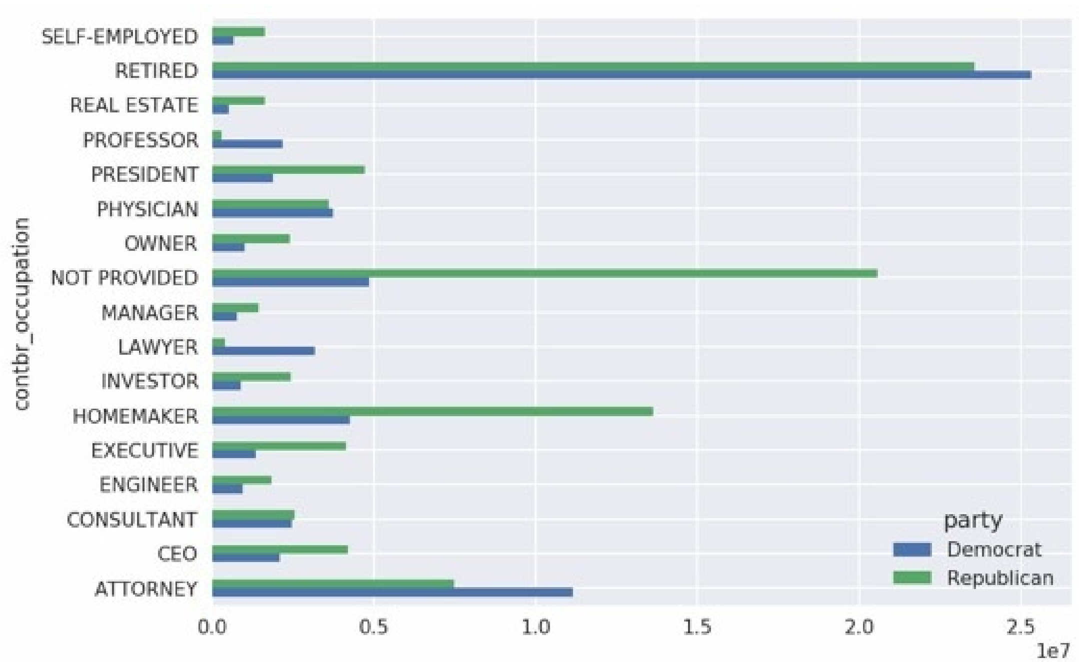


图14-12 对各党派总出资额最高的职业

你可能还想了解一下对Obama和Romney总出资额最高的职业和企业。为此，我们先对候选人进行分组，然后使用本章前面介绍的类似top的方法：

```
def get_top_amounts(group, key, n=5):
    totals = group.groupby(key)['contb_receipt_amt'].
    return totals.nlargest(n)
```

然后根据职业和雇主进行聚合：

```
In [207]: grouped = fec_mrbo.groupby('cand_nm')

In [208]: grouped.apply(get_top_amounts, 'contbr_occu
Out[208]:
cand_nm        contbr_occupation
Obama, Barack  RETIRED                          25305116.38
               ATTORNEY                         11141982.97
               INFORMATION REQUESTED             4866973.96
               HOMEMAKER                         4248875.80
               PHYSICIAN                         3735124.94
```

```
                                              ...
Romney, Mitt   HOMEMAKER                    8147446.22
               ATTORNEY                     5364718.82
               PRESIDENT                    2491244.89
               EXECUTIVE                    2300947.03
               C.E.O.                       1968386.11
Name: contb_receipt_amt, Length: 14, dtype: float64

In [209]: grouped.apply(get_top_amounts, 'contbr_empl
Out[209]:
cand_nm        contbr_employer
Obama, Barack  RETIRED                     22694358.85
               SELF-EMPLOYED               17080985.96
               NOT EMPLOYED                 8586308.70
               INFORMATION REQUESTED        5053480.37
               HOMEMAKER                    2605408.54
                                              ...
Romney, Mitt   CREDIT SUISSE                 281150.00
               MORGAN STANLEY                267266.00
               GOLDMAN SACH & CO.            238250.00
               BARCLAYS CAPITAL              162750.00
               H.I.G. CAPITAL                139500.00
Name: contb_receipt_amt, Length: 20, dtype: float64
```

对出资额分组

还可以对该数据做另一种非常实用的分析：利用cut函数根据出资额的大小将数据离散化到多个面元中：

```
In [210]: bins = np.array([0, 1, 10, 100, 1000, 10000
   .....:                  100000, 1000000, 10000000]

In [211]: labels = pd.cut(fec_mrbo.contb_receipt_amt,

In [212]: labels
Out[212]:
```

```
411          (10, 100]
412        (100, 1000]
413        (100, 1000]
414          (10, 100]
415          (10, 100]
               ...
701381       (10, 100]
701382     (100, 1000]
701383         (1, 10]
701384       (10, 100]
701385     (100, 1000]
Name: contb_receipt_amt, Length: 694282, dtype: categ
Categories (8, interval[int64]): [(0, 1] < (1, 10] < 
000, 10000] <
                                  (10000, 100000] < (
 10000000]]
```

现在可以根据候选人姓名以及面元标签对奥巴马和罗姆尼数据进行分组，以得到一个柱状图：

```
In [213]: grouped = fec_mrbo.groupby(['cand_nm', labe

In [214]: grouped.size().unstack(0)
Out[214]:
cand_nm                     Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                              493.0          77.0
(1, 10]                           40070.0        3681.0
(10, 100]                        372280.0       31853.0
(100, 1000]                      153991.0       43357.0
(1000, 10000]                     22284.0       26186.0
(10000, 100000]                       2.0           1.0
(100000, 1000000]                     3.0           NaN
(1000000, 10000000]                   4.0           NaN
```

从这个数据中可以看出，在小额赞助方面，Obama获得的数量

比Romney多得多。你还可以对出资额求和并在面元内规格化，以便图形化显示两位候选人各种赞助额度的比例（见图14-13）：

```
In [216]: bucket_sums = grouped.contb_receipt_amt.sum

In [217]: normed_sums = bucket_sums.div(bucket_sums.su

In [218]: normed_sums
Out[218]:
cand_nm                    Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                          0.805182      0.194818
(1, 10]                         0.918767      0.081233
(10, 100]                       0.910769      0.089231
(100, 1000]                     0.710176      0.289824
(1000, 10000]                   0.447326      0.552674
(10000, 100000]                 0.823120      0.176880
(100000, 1000000]               1.000000           NaN
(1000000, 10000000]             1.000000           NaN

In [219]: normed_sums[:-2].plot(kind='barh')
```

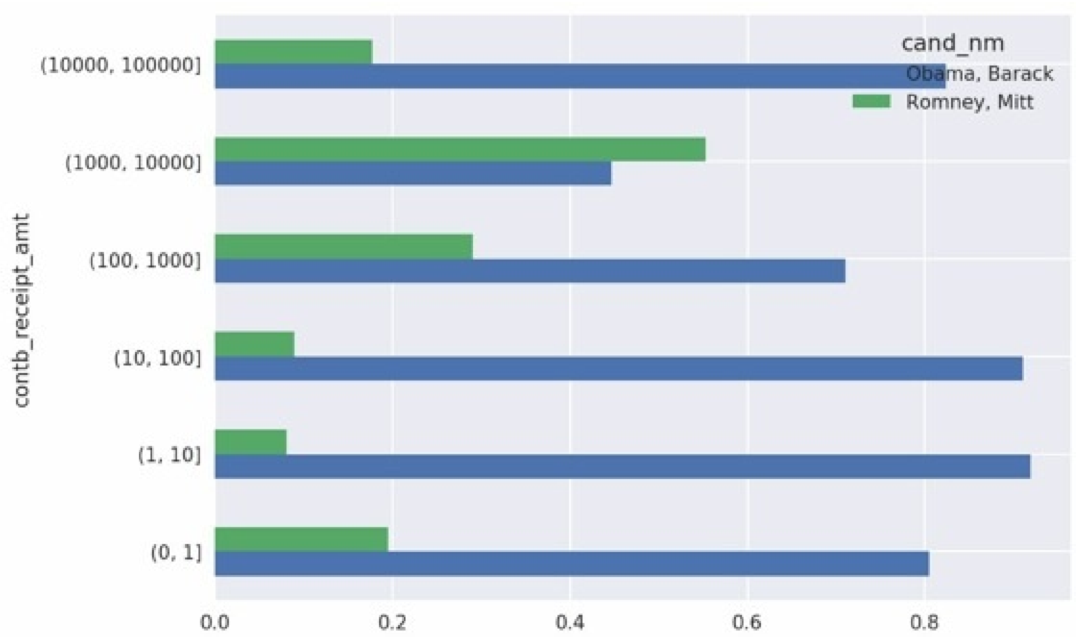


图14-13 两位候选人收到的各种捐赠额度的总额比例

我排除了两个最大的面元，因为这些不是由个人捐赠的。

还可以对该分析过程做许多的提炼和改进。比如说，可以根据赞助人的姓名和邮编对数据进行聚合，以便找出哪些人进行了多次小额捐款，哪些人又进行了一次或多次大额捐款。我强烈建议你下载这些数据并自己摸索一下。

根据州统计赞助信息

根据候选人和州对数据进行聚合是常规操作：

```
In [220]: grouped = fec_mrbo.groupby(['cand_nm', 'con

In [221]: totals = grouped.contb_receipt_amt.sum().un

In [222]: totals = totals[totals.sum(1) > 100000]

In [223]: totals[:10]
Out[223]:
cand_nm     Obama, Barack  Romney, Mitt
contbr_st
AK              281840.15      86204.24
```

```
AL                543123.48     527303.51
AR                359247.28     105556.00
AZ               1506476.98    1888436.23
CA              23824984.24   11237636.60
CO               2132429.49    1506714.12
CT               2068291.26    3499475.45
DC               4373538.80    1025137.50
DE                336669.14      82712.00
FL               7318178.58    8338458.81
```

如果对各行除以总赞助额，就会得到各候选人在各州的总赞助额比例：

```
In [224]: percent = totals.div(totals.sum(1), axis=0)

In [225]: percent[:10]
Out[225]:
cand_nm    Obama, Barack  Romney, Mitt
contbr_st
AK              0.765778      0.234222
AL              0.507390      0.492610
AR              0.772902      0.227098
AZ              0.443745      0.556255
CA              0.679498      0.320502
CO              0.585970      0.414030
CT              0.371476      0.628524
DC              0.810113      0.189887
DE              0.802776      0.197224
FL              0.467417      0.532583
```

14.6 总结

我们已经完成了正文的最后一章。附录中有一些额外的内容，可能对你有用。

本书第一版出版已经有5年了，Python已经成为了一个流行的、

广泛使用的数据分析语言。你从本书中学到的方法，在相当长的一段时间都是可用的。我希望本书介绍的工具和库对你的工作有用。

附录A NumPy高级应用

在这篇附录中，我会深入NumPy库的数组计算。这会包括ndarray更内部的细节，和更高级的数组操作和算法。

这章包括了一些杂乱的章节，不需要仔细研究。

A.1 ndarray对象的内部机理

NumPy的ndarray提供了一种将同质数据块（可以是连续或跨越）解释为多维数组对象的方式。正如你之前所看到的那样，数据类型（dtype）决定了数据的解释方式，比如浮点数、整数、布尔值等。

ndarray如此强大的部分原因是所有数组对象都是数据块的一个跨度视图（strided view）。你可能想知道数组视图arr[::2,::-1]不复制任何数据的原因是什么。简单地说，ndarray不只是一块内存和一个dtype，它还有跨度信息，这使得数组能以各种步幅（step size）在内存中移动。更准确地讲，ndarray内部由以下内容组成：

- 一个指向数据（内存或内存映射文件中的一块数据）的指针。
- 数据类型或dtype，描述在数组中的固定大小值的格子。
- 一个表示数组形状（shape）的元组。
- 一个跨度元组（stride），其中的整数指的是为了前进到当前维度下一个元素需要“跨过”的字节数。

图A-1简单地说明了ndarray的内部结构。

例如，一个10×5的数组，其形状为(10,5)：

```
In [10]: np.ones((10, 5)).shape
Out[10]: (10, 5)
```

一个典型的（C顺序，稍后将详细讲解）3×4×5的float64（8个字节）数组，其跨度为(160,40,8) —— 知道跨度是非常有用的，通

常，跨度在一个轴上越大，沿这个轴进行计算的开销就越大：

```
In [11]: np.ones((3, 4, 5), dtype=np.float64).strides
Out[11]: (160, 40, 8)
```

虽然NumPy用户很少会对数组的跨度信息感兴趣，但它们却是构建非复制式数组视图的重要因素。跨度甚至可以是负数，这样会使数组在内存中后向移动，比如在切片obj[::-1]或obj[:,::-1]中就是这样的。

NumPy数据类型体系

你可能偶尔需要检查数组中所包含的是否是整数、浮点数、字符串或Python对象。因为浮点数的种类很多（从float16到float128），判断dtype是否属于某个大类的工作非常繁琐。幸运的是，dtype都有一个超类（比如np.integer和np.floating），它们可以跟np.issubdtype函数结合使用：

```
In [12]: ints = np.ones(10, dtype=np.uint16)

In [13]: floats = np.ones(10, dtype=np.float32)

In [14]: np.issubdtype(ints.dtype, np.integer)
Out[14]: True

In [15]: np.issubdtype(floats.dtype, np.floating)
Out[15]: True
```

调用dtype的mro方法即可查看其所有的父类：

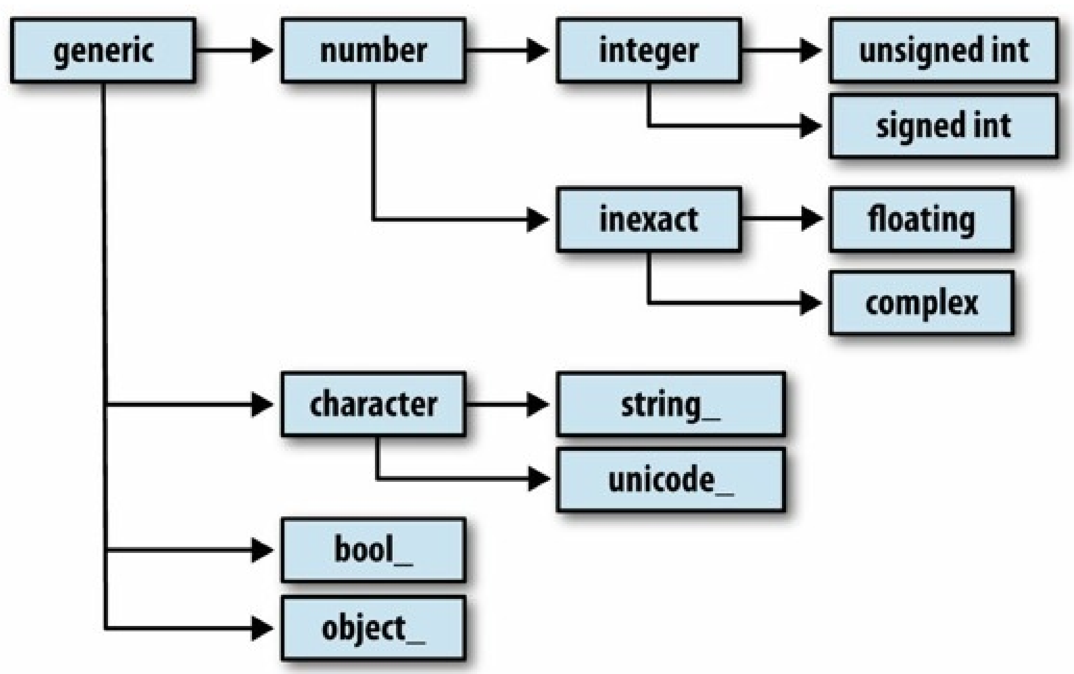
```
In [16]: np.float64.mro()
Out[16]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
```

```
 numpy.generic,
 float,
 object]
```

然后得到:

```
In [17]: np.issubdtype(ints.dtype, np.number)
Out[17]: True
```

大部分NumPy用户完全不需要了解这些知识，但是这些知识偶尔还是能派上用场的。图A-2说明了dtype体系以及父子类关系。



图A-2 NumPy的dtype体系

A.2 高级数组操作

除花式索引、切片、布尔条件取子集等操作之外，数组的操作方式还有很多。虽然pandas中的高级函数可以处理数据分析工作中的许多重型任务，但有时你还是需要编写一些在现有库中找不到的数据算法。

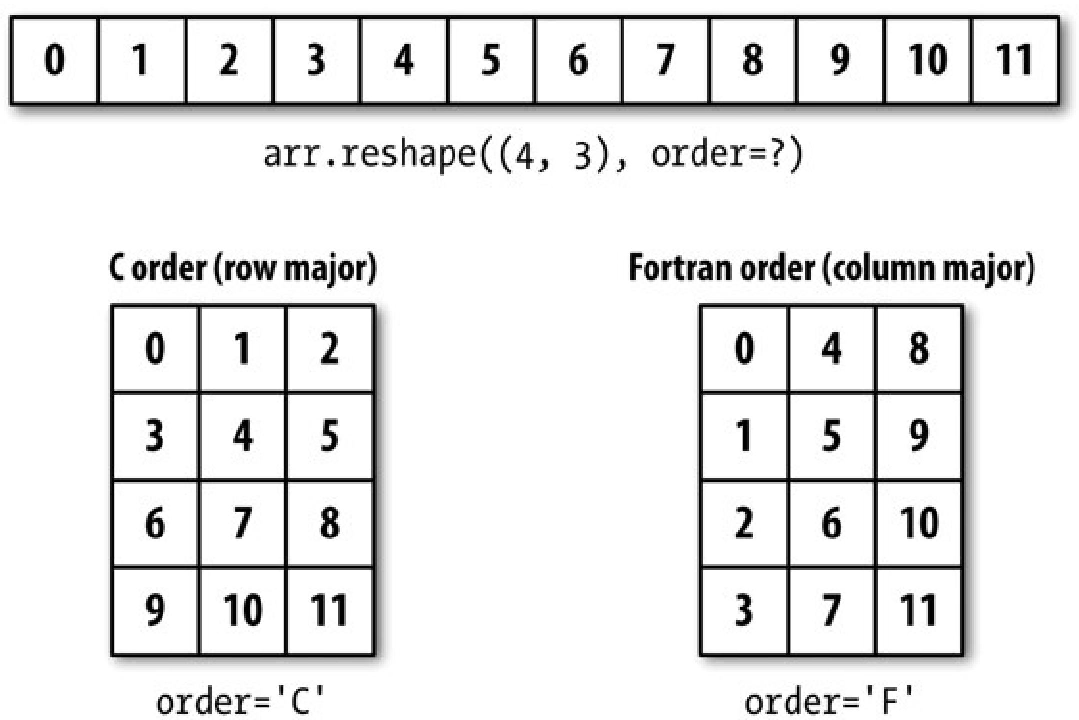
数组重塑

多数情况下，你可以无需复制任何数据，就将数组从一个形状转换为另一个形状。只需向数组的实例方法reshape传入一个表示新形状的元组即可实现该目的。例如，假设有一个一维数组，我们希望将其重新排列为一个矩阵（结果见图A-3）：

```
In [18]: arr = np.arange(8)

In [19]: arr
Out[19]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [20]: arr.reshape((4, 2))
Out[20]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```



图A-3 按C顺序（按行）和按Fortran顺序（按列）进行重塑

多维数组也能被重塑：

```
In [21]: arr.reshape((4, 2)).reshape((2, 4))
Out[21]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

作为参数的形状的其中一维可以是－1，它表示该维度的大小由数据本身推断而来：

```
In [22]: arr = np.arange(15)

In [23]: arr.reshape((5, -1))
Out[23]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

与reshape将一维数组转换为多维数组的运算过程相反的运算通常称为扁平化（flattening）或散开（raveling）：

```
In [27]: arr = np.arange(15).reshape((5, 3))

In [28]: arr
Out[28]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])

In [29]: arr.ravel()
Out[29]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,
```

如果结果中的值与原始数组相同，ravel不会产生源数据的副本。flatten方法的行为类似于ravel，只不过它总是返回数据的副本：

```
In [30]: arr.flatten()
Out[30]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  
```

数组可以被重塑或散开为别的顺序。这对NumPy新手来说是一个比较微妙的问题，所以在下一小节中我们将专门讲解这个问题。

C和Fortran顺序

NumPy允许你更为灵活地控制数据在内存中的布局。默认情况下，NumPy数组是按行优先顺序创建的。在空间方面，这就意味着，对于一个二维数组，每行中的数据项是被存放在相邻内存位置上的。另一种顺序是列优先顺序，它意味着每列中的数据项是被存放在相邻内存位置上的。

由于一些历史原因，行和列优先顺序又分别称为C和Fortran顺序。在FORTRAN 77中，矩阵全都是列优先的。

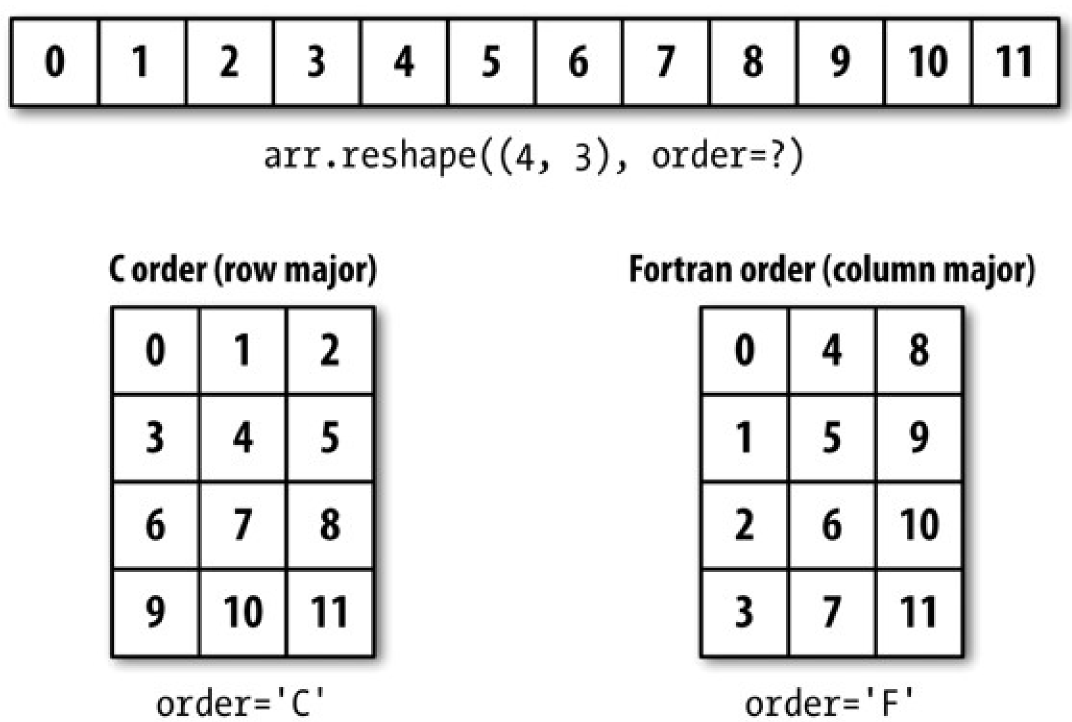
像reshape和reval这样的函数，都可以接受一个表示数组数据存放顺序的order参数。一般可以是'C'或'F'（还有'A'和'K'等不常用的选项，具体请参考NumPy的文档）。图A-3对此进行了说明：

```
In [31]: arr = np.arange(12).reshape((3, 4))

In [32]: arr
Out[32]: 
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [33]: arr.ravel()
Out[33]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  

In [34]: arr.ravel('F')
Out[34]: array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  
```



图A-3 按C（行优先）或Fortran（列优先）顺序进行重塑

二维或更高维数组的重塑过程比较令人费解（见图A-3）。C和Fortran顺序的关键区别就是维度的行进顺序：

- C/行优先顺序：先经过更高的维度（例如，轴1会先于轴0被处理）。
- Fortran/列优先顺序：后经过更高的维度（例如，轴0会先于轴1被处理）。

数组的合并和拆分

numpy.concatenate可以按指定轴将一个由数组组成的序列（如元组、列表等）连接到一起：

```
In [35]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])

In [36]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])

In [37]: np.concatenate([arr1, arr2], axis=0)
Out[37]:
array([[ 1,  2,  3],
```

```
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [38]: np.concatenate([arr1, arr2], axis=1)
Out[38]:
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

对于常见的连接操作，NumPy提供了一些比较方便的方法（如vstack和hstack）。因此，上面的运算还可以表达为：

```
In [39]: np.vstack((arr1, arr2))
Out[39]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [40]: np.hstack((arr1, arr2))
Out[40]:
array([[ 1,  2,  3,  7,  8,  9],
[ 4,  5,  6, 10, 11, 12]])
```

与此相反，split用于将一个数组沿指定轴拆分为多个数组：

```
In [41]: arr = np.random.randn(5, 2)

In [42]: arr
Out[42]:
array([[-0.2047,  0.4789],
       [-0.5194, -0.5557],
       [ 1.9658,  1.3934],
       [ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])

In [43]: first, second, third = np.split(arr, [1, 3])
```

```
In [44]: first
Out[44]: array([[-0.2047,  0.4789]])

In [45]: second
Out[45]:
array([[-0.5194, -0.5557],
       [ 1.9658,  1.3934]])
In [46]: third
Out[46]:
array([[ 0.0929,  0.2817],
       [ 0.769 ,  1.2464]])
```

传入到np.split的值[1,3]指示在哪个索引处分割数组。

表A-1中列出了所有关于数组连接和拆分的函数，其中有些是专门为了方便常见的连接运算而提供的。

函数	说明
concatenate	最一般化的连接，沿一条轴连接一组数组
vstack、row_stack	以面向行的方式对数组进行堆叠（沿轴0）
hstack	以面向列的方式对数组进行堆叠（沿轴1）
column_stack	类似于hstack，但是会先将一维数组转换为二维列向量
dstack	以面向“深度”的方式对数组进行堆叠（沿轴2）
split	沿指定轴在指定的位置拆分数组
hsplit、vsplit、dsplit	split的便捷化函数，分别沿轴0、轴1、轴2进行拆分

表A-1 数组连接函数

堆叠辅助类：r_和c_

NumPy命名空间中有两个特殊的对象——r_和c_，它们可以使数组的堆叠操作更为简洁：

```
In [47]: arr = np.arange(6)

In [48]: arr1 = arr.reshape((3, 2))
```

```
In [49]: arr2 = np.random.randn(3, 2)

In [50]: np.r_[arr1, arr2]
Out[50]:
array([[ 0.    ,  1.    ],
       [ 2.    ,  3.    ],
       [ 4.    ,  5.    ],
       [ 1.0072, -1.2962],
       [ 0.275 ,  0.2289],
       [ 1.3529,  0.8864]])

In [51]: np.c_[np.r_[arr1, arr2], arr]
Out[51]:
array([[ 0.    ,  1.    ,  0.    ],
       [ 2.    ,  3.    ,  1.    ],
       [ 4.    ,  5.    ,  2.    ],
       [ 1.0072, -1.2962,  3.    ],
       [ 0.275 ,  0.2289,  4.    ],
       [ 1.3529,  0.8864,  5.    ]])
```

它还可以将切片转换成数组：

```
In [52]: np.c_[1:6, -10:-5]
Out[52]:
array([[  1, -10],
       [  2,  -9],
       [  3,  -8],
       [  4,  -7],
       [  5,  -6]])
```

r_和c_的具体功能请参考其文档。

元素的重复操作：tile和repeat

对数组进行重复以产生更大数组的工具主要是repeat和tile这两个函数。repeat会将数组中的各个元素重复一定次数，从而产生一

个更大的数组：

```
In [53]: arr = np.arange(3)

In [54]: arr
Out[54]: array([0, 1, 2])

In [55]: arr.repeat(3)
Out[55]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

> 笔记：跟其他流行的数组编程语言（如MATLAB）不同，NumPy中很少需要对数组进行重复（replicate）。这主要是因为广播（broadcasting，我们将在下一节中讲解该技术）能更好地满足该需求。

默认情况下，如果传入的是一个整数，则各元素就都会重复那么多次。如果传入的是一组整数，则各元素就可以重复不同的次数：

```
In [56]: arr.repeat([2, 3, 4])
Out[56]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

对于多维数组，还可以让它们的元素沿指定轴重复：

```
In [57]: arr = np.random.randn(2, 2)

In [58]: arr
Out[58]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [59]: arr.repeat(2, axis=0)
Out[59]:
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])
```

注意，如果没有设置轴向，则数组会被扁平化，这可能不会是你想要的结果。同样，在对多维进行重复时，也可以传入一组整数，这样就会使各切片重复不同的次数：

```
In [60]: arr.repeat([2, 3], axis=0)
Out[60]:
array([[-2.0016, -0.3718],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386],
       [ 1.669 , -0.4386]])

In [61]: arr.repeat([2, 3], axis=1)
Out[61]:
array([[-2.0016, -2.0016, -0.3718, -0.3718, -0.3718],
       [ 1.669 ,  1.669 , -0.4386, -0.4386, -0.4386]]
```

tile的功能是沿指定轴向堆叠数组的副本。你可以形象地将其想象成“铺瓷砖”：

```
In [62]: arr
Out[62]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [63]: np.tile(arr, 2)
Out[63]:
array([[-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

第二个参数是瓷砖的数量。对于标量，瓷砖是水平铺设的，而不是垂直铺设。它可以是一个表示“铺设”布局的元组：

```
In [64]: arr
Out[64]:
```

```
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [65]: np.tile(arr, (2, 1))
Out[65]:
array([[-2.0016, -0.3718],
       [ 1.669 , -0.4386],
       [-2.0016, -0.3718],
       [ 1.669 , -0.4386]])

In [66]: np.tile(arr, (3, 2))
Out[66]:
array([[-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386],
       [-2.0016, -0.3718, -2.0016, -0.3718],
       [ 1.669 , -0.4386,  1.669 , -0.4386]])
```

花式索引的等价函数：take和put

在第4章中我们讲过，获取和设置数组子集的一个办法是通过整数数组使用花式索引：

```
In [67]: arr = np.arange(10) * 100

In [68]: inds = [7, 1, 2, 6]

In [69]: arr[inds]
Out[69]: array([700, 100, 200, 600])
```

ndarray还有其它方法用于获取单个轴向上的选区：

```
In [70]: arr.take(inds)
Out[70]: array([700, 100, 200, 600])

In [71]: arr.put(inds, 42)
```

```
In [72]: arr
Out[72]: array([  0,  42,  42, 300, 400, 500,  42,  4

In [73]: arr.put(inds, [40, 41, 42, 43])

In [74]: arr
Out[74]: array([  0,  41,  42, 300, 400, 500,  43,  4
```

要在其它轴上使用take，只需传入axis关键字即可：

```
In [75]: inds = [2, 0, 2, 1]

In [76]: arr = np.random.randn(2, 4)

In [77]: arr
Out[77]:
array([[-0.5397,  0.477 ,  3.2489, -1.0212],
       [-0.5771,  0.1241,  0.3026,  0.5238]])

In [78]: arr.take(inds, axis=1)
Out[78]:
array([[ 3.2489, -0.5397,  3.2489,  0.477 ],
       [ 0.3026, -0.5771,  0.3026,  0.1241]])
```

put不接受axis参数，它只会在数组的扁平化版本（一维，C顺序）上进行索引。因此，在需要用其他轴向的索引设置元素时，最好还是使用花式索引。

A.3 广播

广播（broadcasting）指的是不同形状的数组之间的算术运算的执行方式。它是一种非常强大的功能，但也容易令人误解，即使是经验丰富的老手也是如此。将标量值跟数组合并时就会发生最简单的广播：

```
In [79]: arr = np.arange(5)

In [80]: arr
Out[80]: array([0, 1, 2, 3, 4])

In [81]: arr * 4
Out[81]: array([ 0,  4,  8, 12, 16])
```

这里我们说：在这个乘法运算中，标量值4被广播到了其他所有的元素上。

看一个例子，我们可以通过减去列平均值的方式对数组的每一列进行距平化处理。这个问题解决起来非常简单：

```
In [82]: arr = np.random.randn(4, 3)

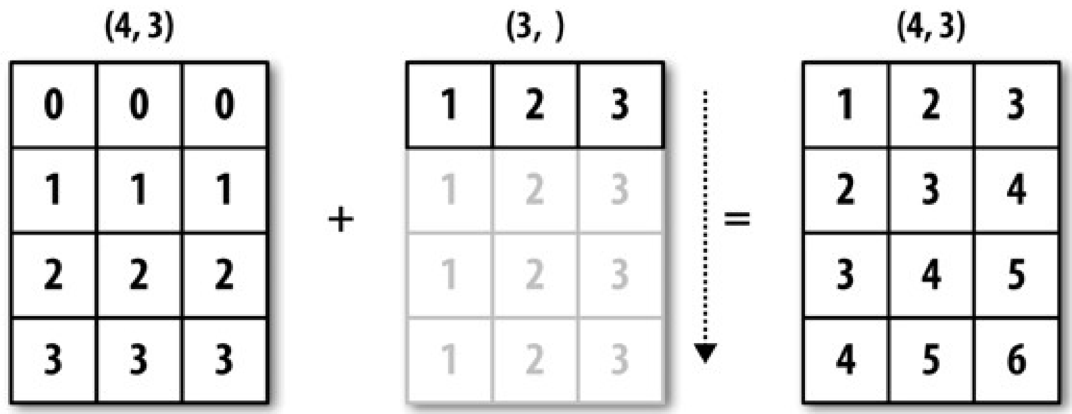
In [83]: arr.mean(0)
Out[83]: array([-0.3928, -0.3824, -0.8768])

In [84]: demeaned = arr - arr.mean(0)

In [85]: demeaned
Out[85]:
array([[ 0.3937,  1.7263,  0.1633],
       [-0.4384, -1.9878, -0.9839],
       [-0.468 ,  0.9426, -0.3891],
       [ 0.5126, -0.6811,  1.2097]])

In [86]: demeaned.mean(0)
Out[86]: array([-0.,  0., -0.])
```

图A-4形象地展示了该过程。用广播的方式对行进行距平化处理会稍微麻烦一些。幸运的是，只要遵循一定的规则，低维度的值是可以被广播到数组的任意维度的（比如对二维数组各列减去行平均值）。



图A-4 一维数组在轴0上的广播

于是就得到了：

> **广播的原则**
>
> 如果两个数组的后缘维度（trailing dimension，即从末尾开始算起的维度）的轴长度相符或其中一方的长度为1，则认为它们是广播兼容的。广播会在缺失和（或）长度为1的维度上进行。

虽然我是一名经验丰富的NumPy老手，但经常还是得停下来画张图并想想广播的原则。再来看一下最后那个例子，假设你希望对各行减去那个平均值。由于arr.mean(0)的长度为3，所以它可以在0轴向上进行广播：因为arr的后缘维度是3，所以它们是兼容的。根据该原则，要在1轴向上做减法（即各行减去行平均值），较小的那个数组的形状必须是(4,1)：

```
In [87]: arr
Out[87]:
array([[ 0.0009,  1.3438, -0.7135],
       [-0.8312, -2.3702, -1.8608],
       [-0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329]])

In [88]: row_means = arr.mean(1)

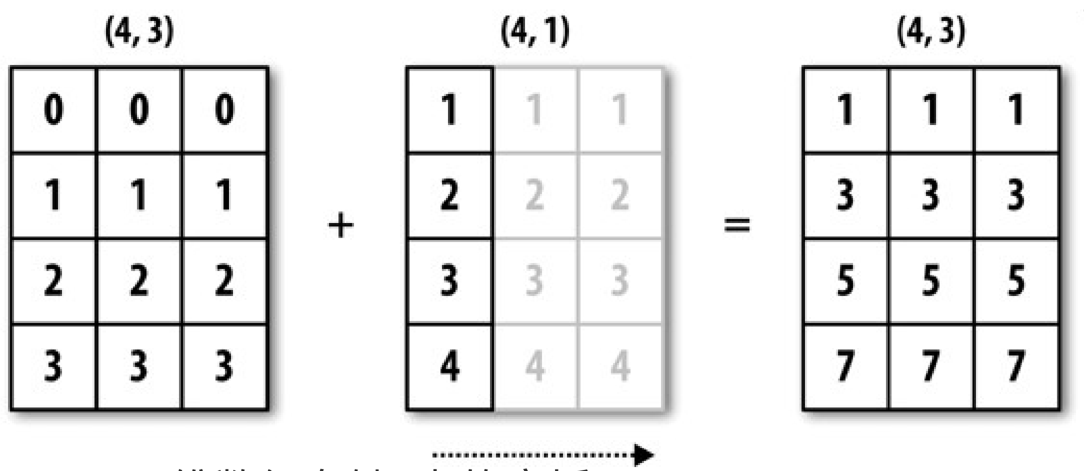
In [89]: row_means.shape
Out[89]: (4,)
```

```
In [90]: row_means.reshape((4, 1))
Out[90]:
array([[ 0.2104],
       [-1.6874],
       [-0.5222],
       [-0.2036]])

In [91]: demeaned = arr - row_means.reshape((4, 1))

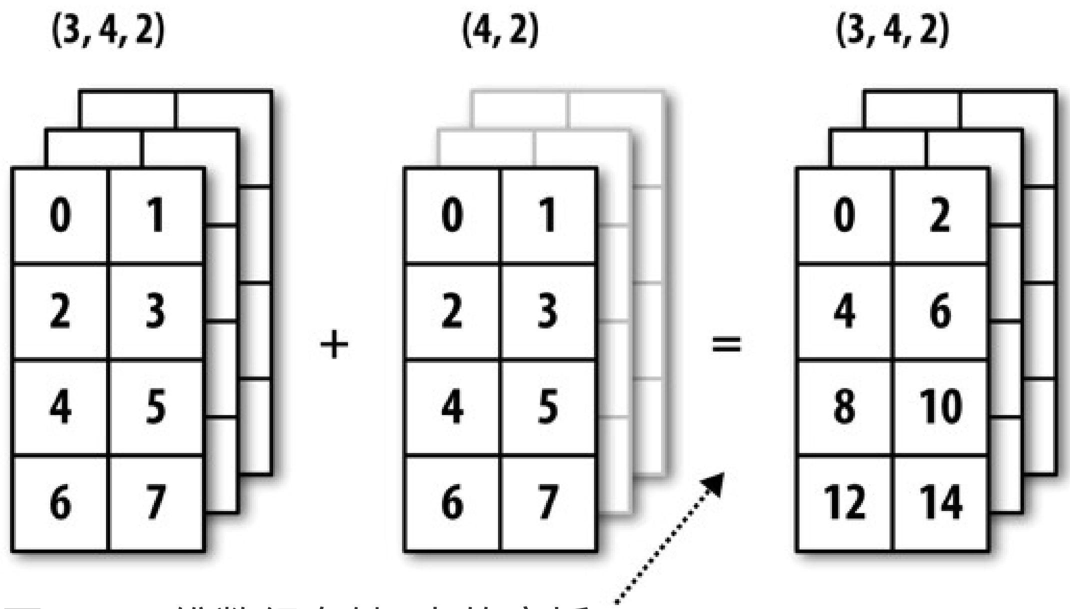
In [92]: demeaned.mean(1)
Out[92]: array([ 0., -0.,  0.,  0.])
```

图A-5说明了该运算的过程。



图A-5 二维数组在轴1上的广播

图A-6展示了另外一种情况，这次是在一个三维数组上沿0轴向加上一个二维数组。



图A-6 三维数组在轴0上的广播

沿其它轴向广播

高维度数组的广播似乎更难以理解，而实际上它也是遵循广播原则的。如果不然，你就会得到下面这样一个错误：

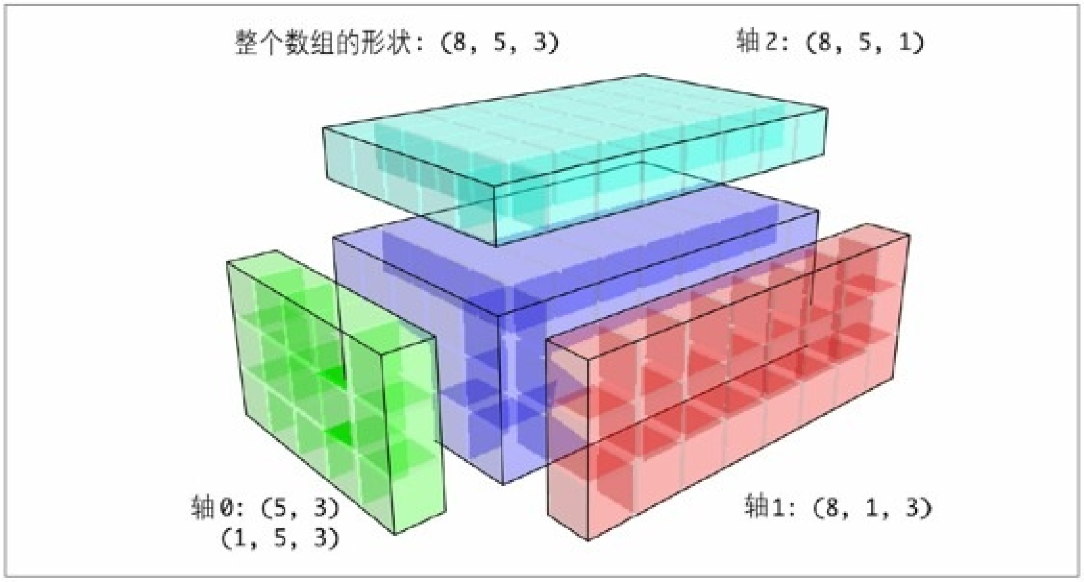
```
In [93]: arr - arr.mean(1)
------------------------------------------------------------
ValueError                                Traceback (
<ipython-input-93-7b87b85a20b2> in <module>()
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together
```

人们经常需要通过算术运算过程将较低维度的数组在除0轴以外的其他轴向上广播。根据广播的原则，较小数组的“广播维”必须为1。在上面那个行距平化的例子中，这就意味着要将行平均值的形状变成(4,1)而不是(4,)：

```
In [94]: arr - arr.mean(1).reshape((4, 1))
Out[94]:
array([[-0.2095,  1.1334, -0.9239],
       [ 0.8562, -0.6828, -0.1734],
```

```
       [-0.3386,  1.0823, -0.7438],
       [ 0.3234, -0.8599,  0.5365]])
```

对于三维的情况，在三维中的任何一维上广播其实也就是将数据重塑为兼容的形状而已。图A-7说明了要在三维数组各维度上广播的形状需求。



图A-7：能在该三维数组上广播的二维数组的形状

于是就有了一个非常普遍的问题（尤其是在通用算法中），即专门为了广播而添加一个长度为1的新轴。虽然reshape是一个办法，但插入轴需要构造一个表示新形状的元组。这是一个很郁闷的过程。因此，NumPy数组提供了一种通过索引机制插入轴的特殊语法。下面这段代码通过特殊的np.newaxis属性以及“全”切片来插入新轴：

```
In [95]: arr = np.zeros((4, 4))

In [96]: arr_3d = arr[:, np.newaxis, :]

In [97]: arr_3d.shape
Out[97]: (4, 1, 4)

In [98]: arr_1d = np.random.normal(size=3)
```

```
In [99]: arr_1d[:, np.newaxis]
Out[99]:
array([[-2.3594],
       [-0.1995],
       [-1.542 ]])

In [100]: arr_1d[np.newaxis, :]
Out[100]: array([[-2.3594, -0.1995, -1.542 ]])
```

因此，如果我们有一个三维数组，并希望对轴2进行距平化，那么只需要编写下面这样的代码就可以了：

```
In [101]: arr = np.random.randn(3, 4, 5)

In [102]: depth_means = arr.mean(2)

In [103]: depth_means
Out[103]:
array([[-0.4735,  0.3971, -0.0228,  0.2001],
       [-0.3521, -0.281 , -0.071 , -0.1586],
       [ 0.6245,  0.6047,  0.4396, -0.2846]])

In [104]: depth_means.shape
Out[104]: (3, 4)

In [105]: demeaned = arr - depth_means[:, :, np.newax

In [106]: demeaned.mean(2)
Out[106]:
array([[ 0.,  0., -0., -0.],
       [ 0.,  0., -0.,  0.],
       [ 0.,  0., -0., -0.]])
```

有些读者可能会想，在对指定轴进行距平化时，有没有一种既通用又不牺牲性能的方法呢？实际上是有的，但需要一些索引方面

的技巧：

```
def demean_axis(arr, axis=0):
    means = arr.mean(axis)

    # This generalizes things like [:, :, np.newaxis]
    indexer = [slice(None)] * arr.ndim
    indexer[axis] = np.newaxis
    return arr - means[indexer]
```

通过广播设置数组的值

算术运算所遵循的广播原则同样也适用于通过索引机制设置数组值的操作。对于最简单的情况，我们可以这样做：

```
In [107]: arr = np.zeros((4, 3))

In [108]: arr[:] = 5

In [109]: arr
Out[109]:
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

但是，假设我们想要用一个一维数组来设置目标数组的各列，只要保证形状兼容就可以了：

```
In [110]: col = np.array([1.28, -0.42, 0.44, 1.6])
In [111]: arr[:] = col[:, np.newaxis]

In [112]: arr
Out[112]:
array([[ 1.28,  1.28,  1.28],
       [-0.42, -0.42, -0.42],
```

```
       [ 0.44,  0.44,  0.44],
       [ 1.6 ,  1.6 ,  1.6 ]])

In [113]: arr[:2] = [[-1.37], [0.509]]

In [114]: arr
Out[114]:
array([[-1.37 , -1.37 , -1.37 ],
       [ 0.509,  0.509,  0.509],
       [ 0.44 ,  0.44 ,  0.44 ],
       [ 1.6  ,  1.6  ,  1.6  ]])
```

A.4 ufunc高级应用

虽然许多NumPy用户只会用到通用函数所提供的快速的元素级运算，但通用函数实际上还有一些高级用法能使我们丢开循环而编写出更为简洁的代码。

ufunc实例方法

NumPy的各个二元ufunc都有一些用于执行特定矢量化运算的特殊方法。表A-2汇总了这些方法，下面我将通过几个具体的例子对它们进行说明。

reduce接受一个数组参数，并通过一系列的二元运算对其值进行聚合（可指明轴向）。例如，我们可以用np.add.reduce对数组中各个元素进行求和：

```
In [115]: arr = np.arange(10)

In [116]: np.add.reduce(arr)
Out[116]: 45

In [117]: arr.sum()
Out[117]: 45
```

起始值取决于ufunc（对于add的情况，就是0）。如果设置了轴

号，约简运算就会沿该轴向执行。这就使你能用一种比较简洁的方式得到某些问题的答案。在下面这个例子中，我们用np.logical_and检查数组各行中的值是否是有序的：

```
In [118]: np.random.seed(12346)  # for reproducibilit

In [119]: arr = np.random.randn(5, 5)

In [120]: arr[::2].sort(1) # sort a few rows

In [121]: arr[:, :-1] < arr[:, 1:]
Out[121]:
array([[ True,  True,  True,  True],
       [False,  True, False, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)

In [122]: np.logical_and.reduce(arr[:, :-1] < arr[:, 
Out[122]: array([ True, False,  True, False,  True], 
```

注意，logical_and.reduce跟all方法是等价的。

ccumulate跟reduce的关系就像cumsum跟sum的关系那样。它产生一个跟原数组大小相同的中间“累计”值数组：

```
In [123]: arr = np.arange(15).reshape((3, 5))

In [124]: np.add.accumulate(arr, axis=1)
Out[124]:
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]])
```

outer用于计算两个数组的叉积：

```
In [125]: arr = np.arange(3).repeat([1, 2, 2])

In [126]: arr
Out[126]: array([0, 1, 1, 2, 2])

In [127]: np.multiply.outer(arr, np.arange(5))
Out[127]:
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8],
       [0, 2, 4, 6, 8]])
```

outer输出结果的维度是两个输入数据的维度之和：

```
In [128]: x, y = np.random.randn(3, 4), np.random.ran

In [129]: result = np.subtract.outer(x, y)

In [130]: result.shape
Out[130]: (3, 4, 5)
```

最后一个方法reduceat用于计算“局部约简”，其实就是一个对数据各切片进行聚合的groupby运算。它接受一组用于指示如何对值进行拆分和聚合的“面元边界”：

```
In [131]: arr = np.arange(10)

In [132]: np.add.reduceat(arr, [0, 5, 8])
Out[132]: array([10, 18, 17])
```

最终结果是在arr[0:5]、arr[5:8]以及arr[8:]上执行的约简。跟其他方法一样，这里也可以传入一个axis参数：

```
In [133]: arr = np.multiply.outer(np.arange(4), np.ar
```

```
In [134]: arr
Out[134]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12]])

In [135]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[135]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]])
```

表A-2总结了部分的ufunc方法。

方法	说明
reduce(x)	通过连续执行原始运算的方式对值进行聚合
accumulate(x)	聚合值，保留所有局部聚合结果
reduceat(x, bins)	“局部”约简（也就是groupby）。约简数据的各个切片以产生聚合型数组
outer(x, y)	对x和y中的每对元素应用原始运算。结果数组的形状为x.shape + y.shape

表A ufunc方法

编写新的ufunc

有多种方法可以让你编写自己的NumPy ufuncs。最常见的是使用NumPy C API，但它超越了本书的范围。在本节，我们讲纯粹的Python ufunc。

numpy.frompyfunc接受一个Python函数以及两个分别表示输入输出参数数量的参数。例如，下面是一个能够实现元素级加法的简单函数：

```
In [136]: def add_elements(x, y):
```

```
   .....:     return x + y

In [137]: add_them = np.frompyfunc(add_elements, 2, 1

In [138]: add_them(np.arange(8), np.arange(8))
Out[138]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=ob
```

用frompyfunc创建的函数总是返回Python对象数组，这一点很不方便。幸运的是，还有另一个办法，即numpy.vectorize。虽然没有frompyfunc那么强大，但可以让你指定输出类型：

```
In [139]: add_them = np.vectorize(add_elements, otype

In [140]: add_them(np.arange(8), np.arange(8))
Out[140]: array([  0.,   2.,   4.,   6.,   8.,  10.,
```

虽然这两个函数提供了一种创建ufunc型函数的手段，但它们非常慢，因为它们在计算每个元素时都要执行一次Python函数调用，这就会比NumPy自带的基于C的ufunc慢很多：

```
In [141]: arr = np.random.randn(10000)

In [142]: %timeit add_them(arr, arr)
4.12 ms +- 182 us per loop (mean +- std. dev. of 7 ru

In [143]: %timeit np.add(arr, arr)
6.89 us +- 504 ns per loop (mean +- std. dev. of 7 ru
```

本章的后面，我会介绍使用Numba（http://numba.pydata.org/），创建快速Python ufuncs。

A.5 结构化和记录式数组

你可能已经注意到了，到目前为止我们所讨论的ndarray都是一

种同质数据容器，也就是说，在它所表示的内存块中，各元素占用的字节数相同（具体根据dtype而定）。从表面上看，它似乎不能用于表示异质或表格型的数据。结构化数组是一种特殊的ndarray，其中的各个元素可以被看做C语言中的结构体（struct，这就是“结构化”的由来）或SQL表中带有多个命名字段的行：

```
In [144]: dtype = [('x', np.float64), ('y', np.int32)

In [145]: sarr = np.array([(1.5, 6), (np.pi, -2)], dt

In [146]: sarr
Out[146]:
array([( 1.5   ,  6), ( 3.1416, -2)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

定义结构化dtype（请参考NumPy的在线文档）的方式有很多。最典型的办法是元组列表，各元组的格式为(field_name,field_data_type)。这样，数组的元素就成了元组式的对象，该对象中各个元素可以像字典那样进行访问：

```
In [147]: sarr[0]
Out[147]: ( 1.5, 6)

In [148]: sarr[0]['y']
Out[148]: 6
```

字段名保存在dtype.names属性中。在访问结构化数组的某个字段时，返回的是该数据的视图，所以不会发生数据复制：

```
In [149]: sarr['x']
Out[149]: array([ 1.5   ,  3.1416])
```

嵌套dtype和多维字段

在定义结构化dtype时，你可以再设置一个形状（可以是一个整数，也可以是一个元组）：

```
In [150]: dtype = [('x', np.int64, 3), ('y', np.int32

In [151]: arr = np.zeros(4, dtype=dtype)

In [152]: arr
Out[152]:
array([([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0)
      dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

在这种情况下，各个记录的x字段所表示的是一个长度为3的数组：

```
In [153]: arr[0]['x']
Out[153]: array([0, 0, 0])
```

这样，访问arr['x']即可得到一个二维数组，而不是前面那个例子中的一维数组：

```
In [154]: arr['x']
Out[154]:
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

这就使你能用单个数组的内存块存放复杂的嵌套结构。你还可以嵌套dtype，作出更复杂的结构。下面是一个简单的例子：

```
In [155]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]),

In [156]: data = np.array([((1, 2), 5), ((3, 4), 6)],

In [157]: data['x']
```

```
Out[157]:
array([( 1.,  2.), ( 3.,  4.)],
      dtype=[('a', '<f8'), ('b', '<f4')])

In [158]: data['y']
Out[158]: array([5, 6], dtype=int32)

In [159]: data['x']['a']
Out[159]: array([ 1.,  3.])
```

pandas的DataFrame并不直接支持该功能，但它的分层索引机制跟这个差不多。

为什么要用结构化数组

跟pandas的DataFrame相比，NumPy的结构化数组是一种相对较低级的工具。它可以将单个内存块解释为带有任意复杂嵌套列的表格型结构。由于数组中的每个元素在内存中都被表示为固定的字节数，所以结构化数组能够提供非常快速高效的磁盘数据读写（包括内存映像）、网络传输等功能。

结构化数组的另一个常见用法是，将数据文件写成定长记录字节流，这是C和C++代码中常见的数据序列化手段（业界许多历史系统中都能找得到）。只要知道文件的格式（记录的大小、元素的顺序、字节数以及数据类型等），就可以用np.fromfile将数据读入内存。这种用法超出了本书的范围，知道这点就可以了。

A.6 更多有关排序的话题

跟Python内置的列表一样，ndarray的sort实例方法也是就地排序。也就是说，数组内容的重新排列是不会产生新数组的：

```
In [160]: arr = np.random.randn(6)

In [161]: arr.sort()
```

```
In [162]: arr
Out[162]: array([-1.082 ,  0.3759,  0.8014,  1.1397,
```

在对数组进行就地排序时要注意一点，如果目标数组只是一个视图，则原始数组将会被修改：

```
In [163]: arr = np.random.randn(3, 5)

In [164]: arr
Out[164]:
array([[-0.3318, -1.4711,  0.8705, -0.0847, -1.1329],
       [-1.0111, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]]

In [165]: arr[:, 0].sort()  # Sort first column value

In [166]: arr
Out[166]:
array([[-1.0111, -1.4711,  0.8705, -0.0847, -1.1329],
       [-0.3318, -0.3436,  2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424,  0.8548,  1.038 , -0.329 ]]
```

相反，numpy.sort会为原数组创建一个已排序副本。另外，它所接受的参数（比如kind）跟ndarray.sort一样：

```
In [167]: arr = np.random.randn(5)

In [168]: arr
Out[168]: array([-1.1181, -0.2415, -2.0051,  0.7379,

In [169]: np.sort(arr)
Out[169]: array([-2.0051, -1.1181, -1.0614, -0.2415,

In [170]: arr
Out[170]: array([-1.1181, -0.2415, -2.0051,  0.7379,
```

这两个排序方法都可以接受一个axis参数，以便沿指定轴向对各块数据进行单独排序：

```
In [171]: arr = np.random.randn(3, 5)

In [172]: arr
Out[172]:
array([[ 0.5955, -0.2682,  1.3389, -0.1872,  0.9111],
       [-0.3215,  1.0054, -0.5168,  1.1925, -0.1989],
       [ 0.3969, -1.7638,  0.6071, -0.2222, -0.2171]]

In [173]: arr.sort(axis=1)

In [174]: arr
Out[174]:
array([[-0.2682, -0.1872,  0.5955,  0.9111,  1.3389],
       [-0.5168, -0.3215, -0.1989,  1.0054,  1.1925],
       [-1.7638, -0.2222, -0.2171,  0.3969,  0.6071]]
```

你可能注意到了，这两个排序方法都不可以被设置为降序。其实这也无所谓，因为数组切片会产生视图（也就是说，不会产生副本，也不需要任何其他的计算工作）。许多Python用户都很熟悉一个有关列表的小技巧：values[::-1]可以返回一个反序的列表。对ndarray也是如此：

```
In [175]: arr[:, ::-1]
Out[175]:
array([[ 1.3389,  0.9111,  0.5955, -0.1872, -0.2682],
       [ 1.1925,  1.0054, -0.1989, -0.3215, -0.5168],
       [ 0.6071,  0.3969, -0.2171, -0.2222, -1.7638]]
```

间接排序：argsort和lexsort

在数据分析工作中，常常需要根据一个或多个键对数据集进行排序。例如，一个有关学生信息的数据表可能需要以姓和名进行排

序（先姓后名）。这就是间接排序的一个例子，如果你阅读过有关pandas的章节，那就已经见过不少高级例子了。给定一个或多个键，你就可以得到一个由整数组成的索引数组（我亲切地称之为索引器），其中的索引值说明了数据在新顺序下的位置。argsort和numpy.lexsort就是实现该功能的两个主要方法。下面是一个简单的例子：

```
In [176]: values = np.array([5, 0, 1, 3, 2])

In [177]: indexer = values.argsort()

In [178]: indexer
Out[178]: array([1, 2, 4, 3, 0])

In [179]: values[indexer]
Out[179]: array([0, 1, 2, 3, 5])
```

一个更复杂的例子，下面这段代码根据数组的第一行对其进行排序：

```
In [180]: arr = np.random.randn(3, 5)

In [181]: arr[0] = values

In [182]: arr
Out[182]:
array([[ 5.    ,  0.    ,  1.    ,  3.    ,  2.    ],
       [-0.3636, -0.1378,  2.1777, -0.4728,  0.8356],
       [-0.2089,  0.2316,  0.728 , -1.3918,  1.9956]]

In [183]: arr[:, arr[0].argsort()]
Out[183]:
array([[ 0.    ,  1.    ,  2.    ,  3.    ,  5.    ],
       [-0.1378,  2.1777,  0.8356, -0.4728, -0.3636],
       [ 0.2316,  0.728 ,  1.9956, -1.3918, -0.2089]]
```

lexsort跟argsort差不多，只不过它可以一次性对多个键数组执行间接排序（字典序）。假设我们想对一些以姓和名标识的数据进行排序：

```
In [184]: first_name = np.array(['Bob', 'Jane', 'Stev

In [185]: last_name = np.array(['Jones', 'Arnold', 'A

In [186]: sorter = np.lexsort((first_name, last_name)

In [187]: sorter
Out[187]: array([1, 2, 3, 0, 4])

In [188]: zip(last_name[sorter], first_name[sorter])
Out[188]: <zip at 0x7fa203eda1c8>
```

刚开始使用lexsort的时候可能会比较容易头晕，这是因为键的应用顺序是从最后一个传入的算起的。不难看出，last_name是先于first_name被应用的。

> 笔记：Series和DataFrame的sort_index以及Series的order方法就是通过这些函数的变体（它们还必须考虑缺失值）实现的。

其他排序算法

稳定的（stable）排序算法会保持等价元素的相对位置。对于相对位置具有实际意义的那些间接排序而言，这一点非常重要：

```
In [189]: values = np.array(['2:first', '2:second', '
.....:                    '1:third'])

In [190]: key = np.array([2, 2, 1, 1, 1])

In [191]: indexer = key.argsort(kind='mergesort')
```

```
In [192]: indexer
Out[192]: array([2, 3, 4, 0, 1])

In [193]: values.take(indexer)
Out[193]:
array(['1:first', '1:second', '1:third', '2:first', '
      dtype='<U8')
```

mergesort（合并排序）是唯一的稳定排序，它保证有O(n log n)的性能（空间复杂度），但是其平均性能比默认的quicksort（快速排序）要差。表A-3列出了可用的排序算法及其相关的性能指标。大部分用户完全不需要知道这些东西，但了解一下总是好的。

kind	速度	稳定性	工作空间	最坏的情况
'quicksort'	1	否	0	$O(n^2)$
'mergesort'	2	是	n/2	O(n log n)
'heapsort'	3	否	0	O(n log n)

表A-3 数组排序算法

部分排序数组

排序的目的之一可能是确定数组中最大或最小的元素。NumPy有两个优化方法，numpy.partition和np.argpartition，可以在第k个最小元素划分的数组：

```
In [194]: np.random.seed(12345)

In [195]: arr = np.random.randn(20)

In [196]: arr
Out[196]:
array([-0.2047,  0.4789, -0.5194, -0.5557,  1.9658,
        0.2817,  0.769 ,  1.2464,  1.0072, -1.2962,
        1.3529,  0.8864, -2.0016, -0.3718,  1.669 , -
```

```
In [197]: np.partition(arr, 3)
Out[197]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -(
        0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  (
        1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1
```

当你调用partition(arr, 3)，结果中的头三个元素是最小的三个，没有特定的顺序。numpy.argpartition与numpy.argsort相似，会返回索引，重排数据为等价的顺序：

```
In [198]: indices = np.argpartition(arr, 3)

In [199]: indices
Out[199]:
array([16, 11,  3,  2, 17, 19,  0,  7,  8,  1, 10,  6
        4, 18,  9])

In [200]: arr.take(indices)
Out[200]:
array([-2.0016, -1.2962, -0.5557, -0.5194, -0.3718, -(
        0.2817,  0.769 ,  0.4789,  1.0072,  0.0929,  (
        1.3529,  0.8864,  1.3934,  1.9658,  1.669 ,  1
```

numpy.searchsorted：在有序数组中查找元素

searchsorted是一个在有序数组上执行二分查找的数组方法，只要将值插入到它返回的那个位置就能维持数组的有序性：

```
In [201]: arr = np.array([0, 1, 7, 12, 15])

In [202]: arr.searchsorted(9)
Out[202]: 3
```

你可以传入一组值就能得到一组索引：

```
In [203]: arr.searchsorted([0, 8, 11, 16])
```

```
Out[203]: array([0, 3, 3, 5])
```

从上面的结果中可以看出，对于元素0，searchsorted会返回0。这是因为其默认行为是返回相等值组的左侧索引：

```
In [204]: arr = np.array([0, 0, 0, 1, 1, 1, 1])

In [205]: arr.searchsorted([0, 1])
Out[205]: array([0, 3])

In [206]: arr.searchsorted([0, 1], side='right')
Out[206]: array([3, 7])
```

再来看searchsorted的另一个用法，假设我们有一个数据数组（其中的值在0到10000之间），还有一个表示“面元边界”的数组，我们希望用它将数据数组拆分开：

```
In [207]: data = np.floor(np.random.uniform(0, 10000,

In [208]: bins = np.array([0, 100, 1000, 5000, 10000]

In [209]: data
Out[209]:
array([ 9940.,  6768.,  7908.,  1709.,   268.,  8003.
        4917.,  5262.,  5963.,   519.,  8950.,  7282.
        8101.,   959.,  2189.,  2587.,  4681.,  4593.
        5314.,  1677.,  7688.,  9281.,  6094.,  1501.
        8486.,  9110.,  3838.,  3154.,  5683.,  1878.
        7996.,  5735.,  9732.,  6340.,  8884.,  4954.
        5039.,  2256.])
```

然后，为了得到各数据点所属区间的编号（其中1表示面元[0,100)），我们可以直接使用searchsorted：

```
In [210]: labels = bins.searchsorted(data)
```

```
In [211]: labels
Out[211]:
array([4, 4, 4, 3, 2, 4, 4, 2, 3, 4, 4, 2, 4, 4, 4, 4
       3, 4, 3, 4, 4, 4, 3, 3, 3, 4, 4, 3, 3, 4, 3, 3
       3, 4, 4, 3])
```

通过pandas的groupby使用该结果即可非常轻松地对原数据集进行拆分：

```
In [212]: pd.Series(data).groupby(labels).mean()
Out[212]:
2     498.000000
3    3064.277778
4    7389.035714
dtype: float64
```

A.7 用Numba编写快速NumPy函数

Numba是一个开源项目，它可以利用CPUs、GPUs或其它硬件为类似NumPy的数据创建快速函数。它使用了LLVM项目（http://llvm.org/），将Python代码转换为机器代码。

为了介绍Numba，来考虑一个纯粹的Python函数，它使用for循环计算表达式(x - y).mean()：

```
import numpy as np

def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

这个函数很慢：

```
In [209]: x = np.random.randn(10000000)

In [210]: y = np.random.randn(10000000)

In [211]: %timeit mean_distance(x, y)
1 loop, best of 3: 2 s per loop

In [212]: %timeit (x - y).mean()
100 loops, best of 3: 14.7 ms per loop
```

NumPy的版本要比它快过100倍。我们可以转换这个函数为编译的Numba函数，使用numba.jit函数：

```
In [213]: import numba as nb

In [214]: numba_mean_distance = nb.jit(mean_distance)
```

也可以写成装饰器：

```
@nb.jit
def mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

它要比矢量化的NumPy快：

```
In [215]: %timeit numba_mean_distance(x, y)
100 loops, best of 3: 10.3 ms per loop
```

Numba不能编译Python代码，但它支持纯Python写的一个部分，可以编写数值算法。

Numba是一个深厚的库，支持多种硬件、编译模式和用户插件。它还可以编译NumPy Python API的一部分，而不用for循环。Numba也可以识别可以便以为机器编码的结构体，但是若调用CPython API，它就不知道如何编译。Numba的jit函数有一个选项，nopython=True，它限制了可以被转换为Python代码的代码，这些代码可以编译为LLVM，但没有任何Python C API调用。jit(nopython=True)有一个简短的别名numba.njit。

前面的例子，我们还可以这样写：

```
from numba import float64, njit

@njit(float64(float64[:], float64[:]))
def mean_distance(x, y):
    return (x - y).mean()
```

我建议你学习Numba的线上文档（http://numba.pydata.org/）。下一节介绍一个创建自定义Numpy ufunc对象的例子。

用Numba创建自定义numpy.ufunc对象

numba.vectorize创建了一个编译的NumPy ufunc，它与内置的ufunc很像。考虑一个numpy.add的Python例子：

```
from numba import vectorize

@vectorize
def nb_add(x, y):
    return x + y
```

现在有：

```
In [13]: x = np.arange(10)
```

```
In [14]: nb_add(x, x)
Out[14]: array([  0.,   2.,   4.,   6.,   8.,  10.,  

In [15]: nb_add.accumulate(x, 0)
Out[15]: array([  0.,   1.,   3.,   6.,  10.,  15.,  
```

A.8 高级数组输入输出

我在第4章中讲过，np.save和np.load可用于读写磁盘上以二进制格式存储的数组。其实还有一些工具可用于更为复杂的场景。尤其是内存映像（memory map），它使你能处理在内存中放不下的数据集。

内存映像文件

内存映像文件是一种将磁盘上的非常大的二进制数据文件当做内存中的数组进行处理的方式。NumPy实现了一个类似于ndarray的memmap对象，它允许将大文件分成小段进行读写，而不是一次性将整个数组读入内存。另外，memmap也拥有跟普通数组一样的方法，因此，基本上只要是能用于ndarray的算法就也能用于memmap。

要创建一个内存映像，可以使用函数np.memmap并传入一个文件路径、数据类型、形状以及文件模式：

```
In [214]: mmap = np.memmap('mymmap', dtype='float64',
   .....:                  shape=(10000, 10000))

In [215]: mmap
Out[215]:
memmap([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        ...,
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.],
```

```
        [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

对memmap切片将会返回磁盘上的数据的视图：

```
In [216]: section = mmap[:5]
```

如果将数据赋值给这些视图：数据会先被缓存在内存中（就像是Python的文件对象），调用flush即可将其写入磁盘：

```
In [217]: section[:] = np.random.randn(5, 10000)

In [218]: mmap.flush()

In [219]: mmap
Out[219]:
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.
        [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.
        [-0.1419, -0.3375,  0.4329, ...,  1.2914, -0.
        ...,
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.

In [220]: del mmap
```

只要某个内存映像超出了作用域，它就会被垃圾回收器回收，之前对其所做的任何修改都会被写入磁盘。当打开一个已经存在的内存映像时，仍然需要指明数据类型和形状，因为磁盘上的那个文件只是一块二进制数据而已，没有任何元数据：

```
In [221]: mmap = np.memmap('mymmap', dtype='float64',

In [222]: mmap
Out[222]:
memmap([[ 0.7584, -0.6605,  0.8626, ...,  0.6046, -0.
```

```
        [-1.2113, -1.0375,  0.7093, ..., -1.4117, -0.
        [-0.1419, -0.3375,  0.4329, ...,  1.2914, -0.
        ...,
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.
        [ 0.    ,  0.    ,  0.    , ...,  0.    ,  0.
```

内存映像可以使用前面介绍的结构化或嵌套dtype。

HDF5及其他数组存储方式

PyTables和h5py这两个Python项目可以将NumPy的数组数据存储为高效且可压缩的HDF5格式（HDF意思是“层次化数据格式”）。你可以安全地将好几百GB甚至TB的数据存储为HDF5格式。要学习Python使用HDF5，请参考pandas线上文档。

A.9 性能建议

使用NumPy的代码的性能一般都很不错，因为数组运算一般都比纯Python循环快得多。下面大致列出了一些需要注意的事项：

- 将Python循环和条件逻辑转换为数组运算和布尔数组运算。
- 尽量使用广播。
- 避免复制数据，尽量使用数组视图（即切片）。
- 利用ufunc及其各种方法。

如果单用NumPy无论如何都达不到所需的性能指标，就可以考虑一下用C、Fortran或Cython（等下会稍微介绍一下）来编写代码。我自己在工作中经常会用到Cython（http://cython.org），因为它不用花费我太多精力就能得到C语言那样的性能。

连续内存的重要性

虽然这个话题有点超出本书的范围，但还是要提一下，因为在某些应用场景中，数组的内存布局可以对计算速度造成极大的影响。这是因为性能差别在一定程度上跟CPU的高速缓存

（cache）体系有关。运算过程中访问连续内存块（例如，对以C顺序存储的数组的行求和）一般是最快的，因为内存子系统会将适当的内存块缓存到超高速的L1或L2CPU Cache中。此外，NumPy的C语言基础代码（某些）对连续存储的情况进行了优化处理，这样就能避免一些跨越式的内存访问。

一个数组的内存布局是连续的，就是说元素是以它们在数组中出现的顺序（即Fortran型（列优先）或C型（行优先））存储在内存中的。默认情况下，NumPy数组是以C型连续的方式创建的。列优先的数组（比如C型连续数组的转置）也被称为Fortran型连续。通过ndarray的flags属性即可查看这些信息：

```
In [225]: arr_c = np.ones((1000, 1000), order='C')

In [226]: arr_f = np.ones((1000, 1000), order='F')

In [227]: arr_c.flags

Out[227]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [228]: arr_f.flags
Out[228]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False

In [229]: arr_f.flags.f_contiguous
```

```
Out[229]: True
```

在这个例子中，对两个数组的行进行求和计算，理论上说，arr_c会比arr_f快，因为arr_c的行在内存中是连续的。我们可以在IPython中用%timeit来确认一下：

```
In [230]: %timeit arr_c.sum(1)
784 us +- 10.4 us per loop (mean +- std. dev. of 7 ru

In [231]: %timeit arr_f.sum(1)
934 us +- 29 us per loop (mean +- std. dev. of 7 runs
```

如果想从NumPy中提升性能，这里就应该是下手的地方。如果数组的内存顺序不符合你的要求，使用copy并传入'C'或'F'即可解决该问题：

```
In [232]: arr_f.copy('C').flags
Out[232]:
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  UPDATEIFCOPY : False
```

注意，在构造数组的视图时，其结果不一定是连续的：

```
In [233]: arr_c[:50].flags.contiguous
Out[233]: True

In [234]: arr_c[:, :50].flags
Out[234]:
  C_CONTIGUOUS : False
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
```

```
  ALIGNED : True
  UPDATEIFCOPY : False
```

附录B 更多关于IPython的内容（完）

第2章中，我们学习了IPython shell和Jupyter notebook的基础。本章中，我们会探索IPython更深层次的功能，可以从控制台或在jupyter使用。

B.1 使用命令历史

Ipython维护了一个位于磁盘的小型数据库，用于保存执行的每条指令。它的用途有：

- 只用最少的输入，就能搜索、补全和执行先前运行过的指令；
- 在不同session间保存命令历史；
- 将日志输入/输出历史到一个文件

这些功能在shell中，要比notebook更为有用，因为notebook从设计上是将输入和输出的代码放到每个代码格子中。

搜索和重复使用命令历史

Ipython可以让你搜索和执行之前的代码或其他命令。这个功能非常有用，因为你可能需要重复执行同样的命令，例如%run命令，或其它代码。假设你必须要执行：

```
In[7]: %run first/second/third/data_script.py
```

运行成功，然后检查结果，发现计算有错。解决完问题，然后修改了data_script.py，你就可以输入一些%run命令，然后按Ctrl+P或上箭头。这样就可以搜索历史命令，匹配输入字符的命令。多次按Ctrl+P或上箭头，会继续搜索命令。如果你要执行你想要执行的命令，不要害怕。你可以按下Ctrl-N或下箭头，向前移动历史命令。这样做了几次后，你可以不假思索地按下这些键！

Ctrl-R可以带来如同Unix风格shell（比如bash shell）的readline

的部分增量搜索功能。在Windows上，readline功能是被IPython模仿的。要使用这个功能，先按Ctrl-R，然后输入一些包含于输入行的想要搜索的字符：

```
In [1]: a_command = foo(x, y, z)

(reverse-i-search)`com': a_command = foo(x, y, z)
```

Ctrl-R会循环历史，找到匹配字符的每一行。

输入和输出变量

<textarea class="textarea small-font" id="baidu_translate_input" data-height="70" style="height: 70px; overflow: hidden;"></textarea>

[](javascript:void(0); "添加到收藏夹")

忘记将函数调用的结果分配给变量是非常烦人的。IPython的一个session会在一个特殊变量，存储输入和输出Python对象的引用。前面两个输出会分别存储在（一个下划线）和_（两个下划线）变量：

```
In [24]: 2 ** 27
Out[24]: 134217728

In [25]: _
Out[25]: 134217728
```

输入变量是存储在名字类似_iX的变量中，X是输入行的编号。对于每个输入变量，都有一个对应的输出变量_X。因此在输入第27行之后，会有两个新变量_27 （输出）和_i27（输入）：

```
In [26]: foo = 'bar'

In [27]: foo
Out[27]: 'bar'
```

```
In [28]: _i27
Out[28]: u'foo'

In [29]: _27
Out[29]: 'bar'
```

因为输入变量是字符串，它们可以用Python的exec关键字再次执行：

```
In [30]: exec(_i27)
```

这里，_i27是在In [27]输入的代码。

有几个魔术函数可以让你利用输入和输出历史。%hist可以打印所有或部分的输入历史，加上或不加上编号。%reset可以清理交互命名空间，或输入和输出缓存。%xdel魔术函数可以去除IPython中对一个特别对象的所有引用。对于关于这些魔术方法的更多内容，请查看文档。

> 警告：当处理非常大的数据集时，要记住IPython的输入和输出的历史会造成被引用的对象不被垃圾回收（释放内存），即使你使用del关键字从交互命名空间删除变量。在这种情况下，小心使用xdel %和%reset可以帮助你避免陷入内存问题。

B.2 与操作系统交互

IPython的另一个功能是无缝连接文件系统和操作系统。这意味着，在同时做其它事时，无需退出IPython，就可以像Windows或Unix使用命令行操作，包括shell命令、更改目录、用Python对象（列表或字符串）存储结果。它还有简单的命令别名和目录书签功能。

表B-1总结了调用shell命令的魔术函数和语法。我会在下面几节介绍这些功能。

命令	说明
!cmd	在系统 shell 执行 cmd
output = !cmd args	运行 run，在 output 存储 stdout
%alias alias_name cmd	为一个系统命令定义别名
%bookmark	使用 IPython 的目录书签系统
%cd directory	改变系统的工作目录到传入的目录
%pwd	返回当前的系统工作目录
%pushd directory	将当前目录放在堆栈上，并将其更改为目标目录
%popd	切换到从堆栈顶部弹出的目录
%dirs	返回包含当前目录堆栈的列表
%dhist	打印访问过的目录
%env	返回系统的环境变量的字典
%matplotlib	配置 matplotlib 的选项

表B-1 IPython系统相关命令

Shell命令和别名

用叹号开始一行，是告诉IPython执行叹号后面的所有内容。这意味着你可以删除文件（取决于操作系统，用rm或del）、改变目录或执行任何其他命令。

通过给变量加上叹号，你可以在一个变量中存储命令的控制台输出。例如，在我联网的基于Linux的主机上，我可以获得IP地址为Python变量：

```
In [1]: ip_info = !ifconfig wlan0 | grep "inet "

In [2]: ip_info[0].strip()
Out[2]: 'inet addr:10.0.0.11  Bcast:10.0.0.255  Mask:
```

返回的Python对象ip_info实际上是一个自定义的列表类型，它包含着多种版本的控制台输出。

当使用！，IPython还可以替换定义在当前环境的Python值。要这么做，可以在变量名前面加上$符号：

```
In [3]: foo = 'test*'
```

```
In [4]: !ls $foo
test4.py  test.py  test.xml
```

%alias魔术函数可以自定义shell命令的快捷方式。看一个简单的例子：

```
In [1]: %alias ll ls -l

In [2]: ll /usr
total 332
drwxr-xr-x   2 root root  69632 2012-01-29 20:36 bin/
drwxr-xr-x   2 root root   4096 2010-08-23 12:05 game
drwxr-xr-x 123 root root  20480 2011-12-26 18:08 incl
drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/
drwxr-xr-x  44 root root  69632 2011-12-26 18:08 lib3
lrwxrwxrwx   1 root root      3 2010-08-23 16:02 lib6
drwxr-xr-x  15 root root   4096 2011-10-13 19:03 loca
drwxr-xr-x   2 root root  12288 2012-01-12 09:32 sbin
drwxr-xr-x 387 root root  12288 2011-11-04 22:53 shar
drwxrwsr-x  24 root src    4096 2011-07-17 18:38 src/
```

你可以执行多个命令，就像在命令行中一样，只需用分号隔开：

```
In [558]: %alias test_alias (cd examples; ls; cd ..)

In [559]: test_alias
macrodata.csv  spx.csv  tips.csv
```

当session结束，你定义的别名就会失效。要创建恒久的别名，需要使用配置。

目录书签系统

IPython有一个简单的目录书签系统，可以让你保存常用目录的别名，这样在跳来跳去的时候会非常方便。例如，假设你想创建

一个书签，指向本书的补充内容：

```
In [6]: %bookmark py4da /home/wesm/code/pydata-book
```

这么做之后，当使用%cd魔术命令，就可以使用定义的书签：

```
In [7]: cd py4da
(bookmark:py4da) -> /home/wesm/code/pydata-book
/home/wesm/code/pydata-book
```

如果书签的名字，与当前工作目录的一个目录重名，你可以使用-b标志来覆写，使用书签的位置。使用%bookmark的-l选项，可以列出所有的书签：

```
In [8]: %bookmark -l
Current bookmarks:
py4da -> /home/wesm/code/pydata-book-source
```

书签，和别名不同，在session之间是保持的。

B.3 软件开发工具

除了作为优秀的交互式计算和数据探索环境，IPython也是有效的Python软件开发工具。在数据分析中，最重要的是要有正确的代码。幸运的是，IPython紧密集成了和加强了Python内置的pdb调试器。第二，需要快速的代码。对于这点，IPython有易于使用的代码计时和分析工具。我会详细介绍这些工具。

交互调试器

IPython的调试器用tab补全、语法增强、逐行异常追踪增强了pdb。调试代码的最佳时间就是刚刚发生错误。异常发生之后就输入%debug，就启动了调试器，进入抛出异常的堆栈框架：

```
In [2]: run examples/ipython_bug.py
--------------------------------------------------------------
AssertionError                               Traceback (r
```

```
/home/wesm/code/pydata-book/examples/ipython_bug.py i
     13     throws_an_exception()
     14
---> 15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py i
11 def calling_things():
     12     works_fine()
---> 13     throws_an_exception()
     14
     15 calling_things()

/home/wesm/code/pydata-book/examples/ipython_bug.py i
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:

In [3]: %debug
> /home/wesm/code/pydata-book/examples/ipython_bug.py
      8     b = 6
----> 9     assert(a + b == 10)
     10

ipdb>
```

一旦进入调试器，你就可以执行任意的Python代码，在每个堆栈框架中检查所有的对象和数据（解释器会保持它们活跃）。默认是从错误发生的最低级开始。通过u（up）和d（down），你可以在不同等级的堆栈踪迹切换：

```
ipdb> u
> /home/wesm/code/pydata-book/examples/ipython_bug.py
```

```
     12     works_fine()
---> 13     throws_an_exception()
     14
```

执行%pdb命令，可以在发生任何异常时让IPython自动启动调试器，许多用户会发现这个功能非常好用。

用调试器帮助开发代码也很容易，特别是当你希望设置断点或在函数和脚本间移动，以检查每个阶段的状态。有多种方法可以实现。第一种是使用%run和-d，它会在执行传入脚本的任何代码之前调用调试器。你必须马上按s（step）以进入脚本：

```
In [5]: run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/
NOTE: Enter 'c' at the ipdb>  prompt to start your sc
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
```

然后，你就可以决定如何工作。例如，在前面的异常，我们可以设置一个断点，就在调用works_fine之前，然后运行脚本，在遇到断点时按c（continue）：

```
ipdb> b 12
ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py
     11 def calling_things():
2--> 12     works_fine()
     13     throws_an_exception()
```

这时，你可以step进入works_fine()，或通过按n（next）执行works_fine()，进入下一行：

```
ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py
2    12     works_fine()
---> 13     throws_an_exception()
     14
```

然后，我们可以进入throws_an_exception，到达发生错误的一行，查看变量。注意，调试器的命令是在变量名之前，在变量名前面加叹号！可以查看内容：

```
ipdb> s
--Call--
> /home/wesm/code/pydata-book/examples/ipython_bug.py
      5
----> 6 def throws_an_exception():
      7     a = 5

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py
      6 def throws_an_exception():
----> 7     a = 5
      8     b = 6

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py
      7     a = 5
----> 8     b = 6
      9     assert(a + b == 10)

ipdb> n
> /home/wesm/code/pydata-book/examples/ipython_bug.py
      8     b = 6
----> 9     assert(a + b == 10)
```

```
     10

ipdb> !a
5
ipdb> !b
6
```

提高使用交互式调试器的熟练度需要练习和经验。表B-2,列出了所有调试器命令。如果你习惯了IDE，你可能觉得终端的调试器在一开始会不顺手，但会觉得越来越好用。一些Python的IDEs有很好的GUI调试器，选择顺手的就好。

命令	动作
h(elp)	展示命令列表
help command	展示 command 的文档
c(ontinue)	继续执行程序
q(uit)	退出调试器，不执行更多代码
b(reak) number	在当前文件的第 number 行设置断点
b path/to/file.py:number	在指定文件的第 number 行设置断点
s(tep)	进入函数调用
n(ext)	执行当前行，并前进到同级的下一行
u(p)/d(won)	在函数调用栈向上或向下
a(rgs)	展示当前函数的参数
debug statement	在新的（递归）调试器执行命令 statement
l(ist) statement	在当前栈的级别，展示当前位置和上下文
w(here)	打印完整的栈踪迹和当前位置的上下文

表B-2 IPython调试器命令

使用调试器的其它方式

还有一些其它工作可以用到调试器。第一个是使用特殊的set_trace函数（根据pdb.set_trace命名的），这是一个简装的断点。还有两种方法是你可能想用的（像我一样，将其添加到IPython的配置）：

```
from IPython.core.debugger import Pdb
```

```
def set_trace():
    Pdb(color_scheme='Linux').set_trace(sys._getframe

def debug(f, *args, **kwargs):
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

第一个函数set_trace非常简单。如果你想暂时停下来进行仔细检查（比如发生异常之前），可以在代码的任何位置使用set_trace：

```
In [7]: run examples/ipython_bug.py
> /home/wesm/code/pydata-book/examples/ipython_bug.py
     15     set_trace()
---> 16     throws_an_exception()
     17
```

按c（continue）可以让代码继续正常行进。

我们刚看的debug函数，可以让你方便的在调用任何函数时使用调试器。假设我们写了一个下面的函数，想逐步分析它的逻辑：

```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

普通地使用f，就会像f(1, 2, z=3)。而要想进入f，将f作为第一个参数传递给debug，再将位置和关键词参数传递给f：

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
      1 def f(x, y, z):
----> 2     tmp = x + y
      3     return tmp / z

ipdb>
```

这两个简单方法节省了我平时的大量时间。

最后，调试器可以和%run一起使用。脚本通过运行%run -d，就可以直接进入调试器，随意设置断点并启动脚本：

```
In [1]: %run -d examples/ipython_bug.py
Breakpoint 1 at /home/wesm/code/pydata-book/examples/
NOTE: Enter 'c' at the ipdb>  prompt to start your sc
> <string>(1)<module>()

ipdb>
```

加上-b和行号，可以预设一个断点：

```
In [2]: %run -d -b2 examples/ipython_bug.py

Breakpoint 1 at /home/wesm/code/pydata-book/examples/
NOTE: Enter 'c' at the ipdb>  prompt to start your sc
> <string>(1)<module>()

ipdb> c
> /home/wesm/code/pydata-book/examples/ipython_bug.py
      1 def works_fine():
1---> 2     a = 5
      3     b = 6

ipdb>
```

代码计时：%time 和 %timeit

对于大型和长时间运行的数据分析应用，你可能希望测量不同组件或单独函数调用语句的执行时间。你可能想知道哪个函数占用的时间最长。幸运的是，IPython可以让你开发和测试代码时，很容易地获得这些信息。

手动用time模块和它的函数time.clock和time.time给代码计时，既单调又重复，因为必须要写一些无趣的模板化代码：

```
import time
start = time.time()
for i in range(iterations):
    # some code to run here
elapsed_per = (time.time() - start) / iterations
```

因为这是一个很普通的操作，IPython有两个魔术函数，%time和%timeit，可以自动化这个过程。

%time会运行一次语句，报告总共的执行时间。假设我们有一个大的字符串列表，我们想比较不同的可以挑选出特定开头字符串的方法。这里有一个含有600000字符串的列表，和两个方法，用以选出foo开头的字符串：

```
# a very large list of strings
strings = ['foo', 'foobar', 'baz', 'qux',
           'python', 'Guido Van Rossum'] * 100000

method1 = [x for x in strings if x.startswith('foo')]

method2 = [x for x in strings if x[:3] == 'foo']
```

看起来它们的性能应该是同级别的，但事实呢？用%time进行一下测量：

```
In [561]: %time method1 = [x for x in strings if x.st
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
Wall time: 0.19 s

In [562]: %time method2 = [x for x in strings if x[:3
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
```

Wall time（wall-clock time的简写）是主要关注的。第一个方法是第二个方法的两倍多，但是这种测量方法并不准确。如果用%time多次测量，你就会发现结果是变化的。要想更准确，可以使用%timeit魔术函数。给出任意一条语句，它能多次运行这条语句以得到一个更为准确的时间：

```
In [563]: %timeit [x for x in strings if x.startswith
10 loops, best of 3: 159 ms per loop

In [564]: %timeit [x for x in strings if x[:3] == 'foo
10 loops, best of 3: 59.3 ms per loop
```

这个例子说明了解Python标准库、NumPy、pandas和其它库的性能是很有价值的。在大型数据分析中，这些毫秒的时间就会累积起来！

%timeit特别适合分析执行时间短的语句和函数，即使是微秒或纳秒。这些时间可能看起来毫不重要，但是一个20微秒的函数执行1百万次就比一个5微秒的函数长15秒。在上一个例子中，我们可以直接比较两个字符串操作，以了解它们的性能特点：

```
In [565]: x = 'foobar'

In [566]: y = 'foo'

In [567]: %timeit x.startswith(y)
1000000 loops, best of 3: 267 ns per loop

In [568]: %timeit x[:3] == y
10000000 loops, best of 3: 147 ns per loop
```

基础分析：%prun和%run -p

分析代码与代码计时关系很紧密，除了它关注的是“时间花在了哪里”。Python主要的分析工具是cProfile模块，它并不局限于

IPython。cProfile会执行一个程序或任意的代码块，并会跟踪每个函数执行的时间。

使用cProfile的通常方式是在命令行中运行一整段程序，输出每个函数的累积时间。假设我们有一个简单的在循环中进行线型代数运算的脚本（计算一系列的100×100矩阵的最大绝对特征值）：

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in xrange(niter):
        mat = np.random.randn(K, K)
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)
    return results
some_results = run_experiment()
print 'Largest one we saw: %s' % np.max(some_results)
```

你可以用cProfile运行这个脚本，使用下面的命令行：

```
python -m cProfile cprof_example.py
```

运行之后，你会发现输出是按函数名排序的。这样要看出谁耗费的时间多有点困难，最好用-s指定排序：

```
$ python -m cProfile -s cumulative cprof_example.py
Largest one we saw: 11.923204422
    15116 function calls (14927 primitive calls) in 0

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:l
```

```
      1    0.001    0.001    0.721    0.721 cprof_exam
    100    0.003    0.000    0.586    0.006 linalg.py:
    200    0.572    0.003    0.572    0.003 {numpy.lin
      1    0.002    0.002    0.075    0.075 __init__.p
    100    0.059    0.001    0.059    0.001 {method 'r
      1    0.000    0.000    0.044    0.044 add_newdoc
      2    0.001    0.001    0.037    0.019 __init__.p
      2    0.003    0.002    0.030    0.015 __init__.p
      1    0.000    0.000    0.030    0.030 type_check
      1    0.001    0.001    0.021    0.021 __init__.p
      1    0.013    0.013    0.013    0.013 numeric.py
      1    0.000    0.000    0.009    0.009 __init__.p
      1    0.001    0.001    0.008    0.008 __init__.p
    262    0.005    0.000    0.007    0.000 function_b
    100    0.003    0.000    0.005    0.000 linalg.py:
```

只显示出前15行。扫描cumtime列，可以容易地看出每个函数用了多少时间。如果一个函数调用了其它函数，计时并不会停止。cProfile会记录每个函数的起始和结束时间，使用它们进行计时。

除了在命令行中使用，cProfile也可以在程序中使用，分析任意代码块，而不必运行新进程。Ipython的%prun和%run -p，有便捷的接口实现这个功能。%prun使用类似cProfile的命令行选项，但是可以分析任意Python语句，而不用整个py文件：

```
In [4]: %prun -l 7 -s cumulative run_experiment()
         4203 function calls in 0.643 seconds

Ordered by: cumulative time
List reduced from 32 to 7 due to restriction <7>
ncalls  tottime  percall  cumtime  percall filename:l
     1    0.000    0.000    0.643    0.643 <string>:1
     1    0.001    0.001    0.643    0.643 cprof_exam
   100    0.003    0.000    0.583    0.006 linalg.py:
   200    0.569    0.003    0.569    0.003 {numpy.lin
```

```
    100     0.058     0.001     0.058     0.001 {method 'r
    100     0.003     0.000     0.005     0.000 linalg.py:
    200     0.002     0.000     0.002     0.000 {method 'a
```

相似的，调用`%run -p -s cumulative cprof_example.py`有和命令行相似的作用，只是你不用离开Ipython。

在Jupyter notebook中，你可以使用%%prun魔术方法（两个%）来分析一整段代码。这会弹出一个带有分析输出的独立窗口。便于快速回答一些问题，比如“为什么这段代码用了这么长时间”？

使用IPython或Jupyter，还有一些其它工具可以让分析工作更便于理解。其中之一是SnakeViz（https://github.com/jiffyclub/snakeviz/），它会使用d3.js产生一个分析结果的交互可视化界面。

逐行分析函数

有些情况下，用%prun（或其它基于cProfile的分析方法）得到的信息，不能获得函数执行时间的整个过程，或者结果过于复杂，加上函数名，很难进行解读。对于这种情况，有一个小库叫做line_profiler（可以通过PyPI或包管理工具获得）。它包含IPython插件，可以启用一个新的魔术函数%lprun，可以对一个函数或多个函数进行逐行分析。你可以通过修改IPython配置（查看IPython文档或本章后面的配置小节）加入下面这行，启用这个插件：

```
# A list of dotted module names of IPython extensions
c.TerminalIPythonApp.extensions = ['line_profiler']
```

你还可以运行命令：

```
%load_ext line_profiler
```

line_profiler也可以在程序中使用（查看完整文档），但是在IPython中使用是最为强大的。假设你有一个带有下面代码的模块prof_mod，做一些NumPy数组操作：

```
from numpy.random import randn

def add_and_sum(x, y):
    added = x + y
    summed = added.sum(axis=1)
    return summed

def call_function():
    x = randn(1000, 1000)
    y = randn(1000, 1000)
    return add_and_sum(x, y)
```

如果想了解add_and_sum函数的性能，%prun可以给出下面内容：

```
In [569]: %run prof_mod

In [570]: x = randn(3000, 3000)

In [571]: y = randn(3000, 3000)

In [572]: %prun add_and_sum(x, y)
         4 function calls in 0.049 seconds
   Ordered by: internal time
   ncalls  tottime  percall  cumtime  percall filenam
        1    0.036    0.036    0.046    0.046 prof_mo
        1    0.009    0.009    0.009    0.009 {method
        1    0.003    0.003    0.049    0.049 <string
```

上面的做法启发性不大。激活了IPython插件line_profiler，新的命令%lprun就能用了。使用中的不同点是，我们必须告诉%lprun要分析的函数是哪个。语法是：

```
%lprun -f func1 -f func2 statement_to_profile
```

我们想分析add_and_sum，运行：

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
Line #      Hits         Time  Per Hit   % Time  Line
=====================================================
     3                                           def 
     4         1        36510  36510.0     79.5      
     5         1         9425   9425.0     20.5      
     6         1            1      1.0      0.0      
```

这样就容易诠释了。我们分析了和代码语句中一样的函数。看之前的模块代码，我们可以调用call_function并对它和add_and_sum进行分析，得到一个完整的代码性能概括：

```
In [574]: %lprun -f add_and_sum -f call_function call_
Timer unit: 1e-06 s
File: prof_mod.py
Function: add_and_sum at line 3
Total time: 0.005526 s
Line #      Hits         Time  Per Hit   % Time  Line
=====================================================
     3                                           def 
     4         1         4375   4375.0     79.2      
     5         1         1149   1149.0     20.8      
     6         1            2      2.0      0.0      
File: prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s
Line #      Hits         Time  Per Hit   % Time  Line
=====================================================
```

```
     8                                              def 
     9          1         57169  57169.0      47.2      
    10          1         58304  58304.0      48.2      
    11          1          5543   5543.0       4.6      
```

我的经验是用%prun (cProfile)进行宏观分析，%lprun (line_profiler)做微观分析。最好对这两个工具都了解清楚。

> 笔记：使用%lprun必须要指明函数名的原因是追踪每行的执行时间的损耗过多。追踪无用的函数会显著地改变结果。

B.4 使用IPython高效开发的技巧

方便快捷地写代码、调试和使用是每个人的目标。除了代码风格，流程细节（比如代码重载）也需要一些调整。

因此，这一节的内容更像是门艺术而不是科学，还需要你不断的试验，以达成高效。最终，你要能结构优化代码，并且能省时省力地检查程序或函数的结果。我发现用IPython设计的软件比起命令行，要更适合工作。尤其是当发生错误时，你需要检查自己或别人写的数月或数年前写的代码的错误。

重载模块依赖

在Python中，当你输入imimport some_lib，some_lib中的代码就会被执行，所有的变量、函数和定义的引入，就会被存入到新创建的some_lib模块命名空间。当下一次输入some_lib，就会得到一个已存在的模块命名空间的引用。潜在的问题是当你%run一个脚本，它依赖于另一个模块，而这个模块做过修改，就会产生问题。假设我在test_script.py中有如下代码：

```
import some_lib

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

如果你运行过了%run test_script.py，然后修改了some_lib.py，下一次再执行%run test_script.py，还会得到旧版本的some_lib.py，这是因为Python模块系统的“一次加载”机制。这一点区分了Python和其它数据分析环境，比如MATLAB，它会自动传播代码修改。解决这个问题，有多种方法。第一种是在标准库importlib模块中使用reload函数：

```
import some_lib
import importlib

importlib.reload(some_lib)
```

这可以保证每次运行test_script.py时可以加载最新的some_lib.py。很明显，如果依赖更深，在各处都使用reload是非常麻烦的。对于这个问题，IPython有一个特殊的dreload函数（它不是魔术函数）重载深层的模块。如果我运行过some_lib.py，然后输入dreload(some_lib)，就会尝试重载some_lib和它的依赖。不过，这个方法不适用于所有场景，但比重启IPython强多了。

代码设计技巧

对于这单，没有简单的对策，但是有一些原则，是我在工作中发现很好用的。

保持相关对象和数据活跃

为命令行写一个下面示例中的程序是很少见的：

```
from my_functions import g

def f(x, y):
    return g(x + y)

def main():
    x = 6
```

```
    y = 7.5
    result = x + y

if __name__ == '__main__':
    main()
```

在IPython中运行这个程序会发生问题，你发现是什么了吗？运行之后，任何定义在main函数中的结果和对象都不能在IPython中被访问到。更好的方法是将main中的代码直接在模块的命名空间中执行（或者在`__name__ == '__main__':`中，如果你想让这个模块可以被引用）。这样，当你%rundiamante，就可以查看所有定义在main中的变量。这等价于在Jupyter notebook的代码格中定义一个顶级变量。

扁平优于嵌套

深层嵌套的代码总让我联想到洋葱皮。当测试或调试一个函数时，你需要剥多少层洋葱皮才能到达目标代码呢？“扁平优于嵌套”是Python之禅的一部分，它也适用于交互式代码开发。尽量将函数和类去耦合和模块化，有利于测试（如果你是在写单元测试）、调试和交互式使用。

克服对大文件的恐惧

如果你之前是写JAVA（或者其它类似的语言），你可能被告知要让文件简短。在多数语言中，这都是合理的建议：太长会让人感觉是坏代码，意味着重构和重组是必要的。但是，在用IPython开发时，运行10个相关联的小文件（小于100行），比起两个或三个长文件，会让你更头疼。更少的文件意味着重载更少的模块和更少的编辑时在文件中跳转。我发现维护大模块，每个模块都是紧密组织的，会更实用和Pythonic。经过方案迭代，有时会将大文件分解成小文件。

我不建议极端化这条建议，那样会形成一个单独的超大文件。找到一个合理和直观的大型代码模块库和封装结构往往需要一点工作，但这在团队工作中非常重要。每个模块都应该结构紧密，并

且应该能直观地找到负责每个功能领域功能和类。

B.5 IPython高级功能

要全面地使用IPython系统需要用另一种稍微不同的方式写代码，或深入IPython的配置。

让类是对IPython友好的

IPython会尽可能地在控制台美化展示每个字符串。对于许多对象，比如字典、列表和元组，内置的pprint模块可以用来美化格式。但是，在用户定义的类中，你必自己生成字符串。假设有一个下面的简单的类：

```
class Message:
    def __init__(self, msg):
        self.msg = msg
```

如果这么写，就会发现默认的输出不够美观：

```
In [576]: x = Message('I have a secret')

In [577]: x
Out[577]: <__main__.Message instance at 0x60ebbd8>
```

IPython会接收**repr**魔术方法返回的字符串（通过output = repr(obj)），并在控制台打印出来。因此，我们可以添加一个简单的**repr**方法到前面的类中，以得到一个更有用的输出：

```
class Message:
    def __init__(self, msg):
        self.msg = msg

    def __repr__(self):
        return 'Message: %s' % self.msg
In [579]: x = Message('I have a secret')
```

```
In [580]: x
Out[580]: Message: I have a secret
```

文件和配置

通过扩展配置系统，大多数IPython和Jupyter notebook的外观（颜色、提示符、行间距等等）和动作都是可以配置的。通过配置，你可以做到：

- 改变颜色主题
- 改变输入和输出提示符，或删除输出之后、输入之前的空行
- 执行任意Python语句（例如，引入总是要使用的代码或者每次加载IPython都要运行的内容）
- 启用IPython总是要运行的插件，比如line_profiler中的%lprun魔术函数
- 启用Jupyter插件
- 定义自己的魔术函数或系统别名

IPython的配置存储在特殊的ipython_config.py文件中，它通常是在用户home目录的.ipython/文件夹中。配置是通过一个特殊文件。当你启动IPython，就会默认加载这个存储在profile_default文件夹中的默认文件。因此，在我的Linux系统，完整的IPython配置文件路径是：

```
/home/wesm/.ipython/profile_default/ipython_config.py
```

要启动这个文件，运行下面的命令：

```
ipython profile create
```

这个文件中的内容留给读者自己探索。这个文件有注释，解释了每个配置选项的作用。另一点，可以有多个配置文件。假设你想要另一个IPython配置文件，专门是为另一个应用或项目的。创建一个新的配置文件很简单，如下所示：

```
ipython profile create secret_project
```

做完之后，在新创建的profile_secret_project目录便捷配置文件，然后如下启动IPython：

```
$ ipython --profile=secret_project
Python 3.5.1 | packaged by conda-forge | (default, Ma
Type "copyright", "credits" or "license" for more inf

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's f
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' f

IPython profile: secret_project
```

和之前一样，IPython的文档是一个极好的学习配置文件的资源。

配置Jupyter有些不同，因为你可以使用除了Python的其它语言。要创建一个类似的Jupyter配置文件，运行：

```
jupyter notebook --generate-config
```

这样会在home目录的.jupyter/jupyter_notebook_config.py创建配置文件。编辑完之后，可以将它重命名：

```
$ mv ~/.jupyter/jupyter_notebook_config.py ~/.jupyter
```

打开Jupyter之后，你可以添加--config参数：

```
 jupyter notebook --config=~/.jupyter/my_custom_confi
```

B.6 总结

学习过本书中的代码案例，你的Python技能得到了一定的提升，我建议你持续学习IPython和Jupyter。因为这两个项目的设计初衷就是提高生产率的，你可能还会发现一些工具，可以让你更便捷地使用Python和计算库。

你可以在nbviewer（https://nbviewer.jupyter.org/）上找到更多有趣的Jupyter notebooks。

> 后记：经过三个月，总算翻译完成了这本书。工作砌码，回家码字。最大的改变是，十个手指头，除了两个大拇指和右手的小拇指，其它指尖竟然磨出了茧。读者们持续的阅读、点赞、留言、指出错误，让我感觉是和很多人一起完成一项有意义的事情。Thanks all！